

ПРОГРАММИРОВАНИЕ на C++ глазами ХАКЕРА

2-е издание

+cd

Михаил Фленов

Как сделать код маленьким, а программу невидимой
От простых шуточных программ до сложной манипуляции системой
Примеры работы через компоненты C++ и через Windows Sockets
Как работать с портами компьютера и получать информацию о системе
Интересные алгоритмы написания сетевых программ

bhv

Михаил Фленов

ПРОГРАММИРОВАНИЕ на C++ ГЛАЗАМИ ХАКЕРА

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2009

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1
Ф69

Фленов М. Е.

Ф69 Программирование на C++ глазами хакера. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2009. — 352 с.: ил. + CD-ROM

ISBN 978-5-9775-0303-7

Рассмотрены нестандартные приемы программирования, а также примеры использования недокументированных функций и возможностей языка C++ при разработке шуточных программ и серьезных сетевых приложений для диагностики сетей, управления различными сетевыми устройствами и просто при повседневном использовании интернет-приложений. Во втором издании содержатся новые и переработаны старые примеры, а в качестве среды разработки используется Visual Studio 2008, хотя большинство описываемых примеров работоспособны в более старых версиях и в CodeGear C++ Builder.

Компакт-диск содержит исходные примеры из книги, а также популярные приложения компании CyD Software Labs.

Для программистов

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 24.11.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 28,38.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0303-7

Оглавление

Введение

По своей натуре я заядлый программист и не буду блистать литературными способностями. Зато я постараюсь поделиться своими знаниями и надеюсь рассказать вам что-то новое, раскрыть некоторые из секретов хакеров. Отдельные вещи, которые будут описаны, возможно, вам уже известны, но не все понимают смысл используемого кода. Я же постараюсь описать все как можно подробнее и показать кое-какие известные мне интересные приемы программирования.

Я попытался привести максимальное количество примеров на языке программирования C++. Среди них множество примеров шуточных программ и сетевых приложений. Почему именно на эти две темы я сделаю упор? Если верить моим глазам (а иногда они меня не подводят), именно эти две темы интересует множество читателей, особенно начинающих.

Если вы плохо знакомы с программированием, то лучше начинать изучение с тех примеров и тем, которые представляют наибольший интерес. В этом случае вам будет интереснее постигать новые высоты и получать знания, и обучение будет не в тягость. Чем изучать программирование на скучных задачках, лучше пойти погулять с друзьями.

Чистой теории в книге будет мало, но зато практических занятий — хоть отбавляй. Вы все сможете увидеть своими глазами и сразу же попробовать. Я считаю, что практика является наилучшим учителем. Практические занятия отлагаются в памяти лучше всего. Теория может оказаться бесполезным знанием, если не знать, как применить эту теорию на практике. Поэтому мы будем учиться и узнавать, как решать проблемы.

Если вы ожидали увидеть в данной книге примеры и описания вирусов, то вы сильно ошиблись. Ничего разрушительного мы рассматривать и делать не будем. Я занимаюсь только созиданием. Ни одна из шуточных программ

не будет приводить к печальным последствиям. Шутка хороша, когда она не причиняет боль, а может вызвать у окружающих хотя бы улыбку.

О книге

Для эффективной работы с книгой вам понадобятся минимальные знания C++ и начальные навыки общения с компьютером. Вы должны уметь создать простое приложение, знать, что такое переменные, циклы и как с ними работать. Не помешают знания адресации, указателей и их необходимости. Это позволит вам лучше понимать описываемые примеры. Что касается сетевого программирования, то его я опишу достаточно подробно, начиная с основ и заканчивая сложными примерами. Так что тут начальные знания желательны, но не обязательны.

Я постарался описать все как можно проще, в тексте программ приведено максимум комментариев, чтобы вы наслаждались чтением, и оно не было для вас утомительным.

Эта книга построена не так, как многие другие. В ней нет нудных теоретических рассуждений, а только максимум примеров и исходных кодов. Ее можно воспринимать как практическое руководство к действию. Только на практике вы сможете понять и усвоить все теоретически полученные знания.

Программисты в чем-то похожи на врачей: если врач теоретически знает симптомы болезни, но на практике не может точно отличить отравление от аппендицита, то такого врача лучше не подпускать к больному. Точно так же и программист: если он знает, как функционирует сетевой протокол, но не может с ним работать, то его сетевые программы никогда не будут работать правильно и эффективно.

Ничто не может привести к такому пониманию предмета, как хорошая практика. Когда вы можете "пощупать" все своими руками, то теория запоминается гораздо лучше.

Вот пример из жизни. Я проходил обучение в известном университете на курсах Microsoft по администрированию и программированию сервера баз данных SQL Server. Курсы очень хорошие, и преподаватель старался все очень подробно и доходчиво изложить. Но сам курс был поставлен корпорацией как теоретический, с небольшим добавлением лабораторных работ. В результате нам очень хорошо объяснили, ЧТО может делать сервер. Но когда после курсов я столкнулся с реальной ситуацией, то понял, что не знаю, КАК сделать что-либо. Приходилось снова открывать книгу, которую дали мне в центре обучения (она была на английском языке), и, читая обширный теоретический материал и маленькие лабораторные, разбираться с реальной

задачей. Уж лучше бы я узнал на курсах, как решить проблему, а не что можно теоретически выполнить, потому что такое обучение, по-моему, только пустая трата времени и денег.

Несмотря на это я не противник теории и не пытаюсь сказать, что теория не нужна. Просто нужно описывать, как решить задачу, и рассказывать, зачем мы делаем какие-то определенные действия. После этого, когда вы будете сталкиваться с подобными проблемами, вы уже будете знать, как сделать что-то, чтобы добиться определенного результата.

Именно практических руководств и просто хороших книг с разносторонними примерами не хватает на полках наших магазинов. Я надеюсь, что моя работа хотя бы частично восполнит пробел в этой сфере и поможет вам в последующей работе и решении различных задач программирования.

Итак, книга, несмотря на свое название, сможет пригодиться всем, потому что в ней будут описываться разные программистские приемы, которые используются в каждодневной практике.

Все примеры, которые будут приведены в книге, можно найти на компакт-диске. Тем не менее, я советую все описанное создавать самостоятельно и обращаться к файлам, только если возникли какие-то проблемы, чтобы сравнить и найти ошибку. Когда вы создаете что-то сами, то получаете хорошие практические навыки, а полученная информация откладывается в памяти намного лучше, чем сто прочтенных страниц теории. Это вторая причина, по которой книга построена на основе большого количества примеров и исходных кодов.

Даже если вы решили воспользоваться готовым примером, обязательно досконально разберитесь с ним. Попробуйте изменить какие-то параметры и посмотреть на результат. Попытайтесь модифицировать пример, добавив какие-то дополнительные возможности. Только так вы сможете понять принцип работы используемых функций или алгоритмов.

В качестве среды разработки и для компиляции примеров я использовал Visual Studio 2008, но многие примеры смогут работать даже в Visual Studio 6, не говоря уже о более новых версиях. Единственная проблема, с которой вы можете столкнуться, — файл проекта. Если команды и функции языка C++ изменяются редко (а я постарался не использовать ничего редкого), то формат файла проекта изменился очень сильно и меняется почти в каждой новой версии.

Кто такой хакер? Как им стать?

Прежде чем приступить к практике, я хочу "загрузить" вашу голову небольшим количеством теоретической информации. А именно, прежде чем читать

книгу, вы обязаны знать, кто такой хакер в моем понимании. Если вы будете подразумевать одно, а я другое, то мы не сможем понять друг друга.

В мире полно вопросов, на которые большинство людей не знают правильного ответа. Мало того, люди используют множество слов, даже не догадываясь об их настоящем значении. Например, многие сильно заблуждаются в понимании термина "хакер". Однако каждый вправе считать свое мнение наиболее правильным. И я не буду утверждать, что именно мое мнение единственно верное, но оно отталкивается от первоначального понятия.

Прежде чем начать обсуждать всем известный термин, я должен обратиться к истории и вспомнить, как все начиналось. А начиналось все еще тогда, когда не было даже международной сети Интернет.

Понятие "хакер" зародилось, когда только стала распространяться первая сеть, которая называлась ARPANET. Тогда это понятие обозначало человека, хорошо разбирающегося в компьютерах. Некоторые даже подразумевали под хакером человека, "помешанного" на компьютерах. Понятие ассоциировали со свободным компьютерщиком, человеком, стремящимся к свободе во всем, что касалось его любимой "игрушки". Именно благодаря этому стремлению и тяге к свободному обмену информацией и началось такое бурное развитие Всемирной сети. Именно хакеры помогли развитию Интернета и создали FIDO. Благодаря им появились UNIX-подобные системы с открытым исходным кодом, в которых сейчас работает большое количество серверов.

В те времена еще не было вирусов, и не внедрилась практика взломов сетей или отдельных компьютеров. Образ хакера-взломщика появился немного позже. Но это только образ. Настоящие хакеры никогда не имели никакого отношения к взломам, а если хакер направлял свои действия на разрушение, то это резко осуждалось виртуальным сообществом.

Настоящий хакер — это творец, а не разрушитель. Так как творцов оказалось больше, чем разрушителей, то истинные хакеры выделили тех, кто занимается взломом, в отдельную группу и назвали их крэкерами (взломщиками) или просто вандалами. И хакеры, и взломщики являются героями виртуального мира. И те, и другие борются за свободу доступа к информации. Но только крэкеры, как правило, взламывают сайты, закрытые базы данных и другие источники информации ради собственной наживы, ради денег или минутной славы; такого человека можно назвать только преступником (кем он по закону и является!).

Хакер — это просто гений в некоторой области. В данной книге мы рассматриваем гения в компьютерной сфере, который хорошо знает свой предмет, создает что-то новое и уникальное и при этом помогает другим. Все, кто приписывают этим людям вандализм, сильно ошибаются. Истинные хакеры никогда не используют свои знания во вред другим. Именно к этому я призы-

ваю в данной книге. Только полезная и познавательная информация, которую вы сможете использовать для повышения своего уровня знаний и улучшения качества программирования.

Теперь давайте разберемся, как стать настоящим хакером. Это обсуждение поможет вам больше узнать об этих людях.

- Вы должны знать свой компьютер и научиться эффективно им управлять. А если вы будете еще знать в нем каждую железку, то это только добавит к вашей оценке по "хакерству" большой и жирный плюс.

  Если честно, то еще лет пять назад я знал все железо в компьютерах, разницу в процессорах вплоть до каждого контакта, а сейчас даже не знаю, чем Core2 Duo отличается от своего предшественника. Я просто покупаю ноутбук и даже не пытаюсь думать о том, как работает там процессор, главное, чтобы он работал хорошо и быстро.

  Что я подразумеваю под умением эффективно управлять своим компьютером? Это значит знать все возможные способы выполнения каждого действия и в каждой ситуации уметь использовать наиболее оптимальный из них. В частности, вы должны научиться пользоваться горячими клавишами и не дергать мышь по любому пустяку. Нажатие клавиш выполняется быстрее, чем любое, даже маленькое, перемещение мыши. Просто приучите себя к этому, и вы увидите все прелести работы с клавиатурой. Лично я использую мышь очень редко и стараюсь всегда применять клавиатуру, потому что это намного быстрее.

  Маленький пример на эту тему. Однажды я работал на фирме, где мой начальник всегда копировал и вставлял данные из буфера обмена с помощью кнопок на панели инструментов или команд контекстного меню, которое появляется при щелчке правой кнопкой мыши. Но если вы делаете так же, то, наверно, знаете, что не везде есть кнопки **Копировать**, **Вставить** или такие же пункты в контекстном меню. В таких случаях мой начальник набирает текст вручную. А ведь можно было бы воспользоваться копированием/вставкой с помощью комбинаций горячих клавиш <Ctrl>+<C>/<Ctrl>+<V> или <Ctrl>+<Ins>/<Shift>+<Ins>, которые реализованы практически во всех современных приложениях, на уровне самой ОС и ее компонентов.

  За копирование и вставку в стандартных компонентах Windows (строки ввода, текстовые поля) отвечает сама операционная система, и тут не нужен дополнительный код, чтобы данные действия заработали. Если программист не предусмотрел кнопку, то это не значит, что данного действия нет. Оно есть, но доступно через горячую клавишу.

- Вы должны досконально изучать все, что вам интересно о компьютерах. Если вас интересует графика, то вы должны изучить лучшие графические

пакеты, научиться рисовать в них любые сцены и создавать самые сложные миры. Если вас интересуют сети, то старайтесь узнать о них все. Если вы считаете, что уже знаете все, то купите книгу потолще по данной теме, и вы поймете, что сильно ошибаетесь. Компьютеры — это такая сфера, в которой невозможно знать все!!!

Хакеры — это прежде всего профессионалы в каком-нибудь деле. И это не обязательно должен быть компьютер или какой-то определенный язык программирования. Хакером можно стать в любой области, но я в данной книге буду рассматривать только компьютерных хакеров, или программистов.

- Желательно уметь программировать. Любой хакер должен знать как минимум один язык программирования. А лучше знать даже несколько языков. Лично я для начала рекомендую всем изучить Delphi или C++. Delphi достаточно прост, быстр, эффективен, а главное, это очень мощный язык. C++ — признанный стандарт во всем мире, но немного сложнее в изучении. Но это не значит, что не надо знать другие языки. Вы можете научиться программировать на чем угодно, даже на языке Basic (использовать его не советую, но знать не помешало бы).

  В связи с последними изменениями на рынке средств разработки и непонятной ситуацией с компанией разработчиком Delphi, я все больше начинаю рекомендовать использовать C++.

  Хотя я не очень люблю Visual Basic за его ограниченность, я видел несколько великолепных программ, которые были написаны именно на этом языке. Глядя на них, сразу хочется назвать их автора хакером, потому что это действительно виртуозная и безупречная работа. Создание из ничего чего-то великолепного — как раз и есть искусство хакерства. А Visual Basic .NET уже ничем не отличается по возможностям и мощности от C#.

  Хакер — это человек, который что-то создает. В большинстве случаев это относится к коду, но можно создавать и графику, и музыку. Все это тоже относится к искусству хакера. Но даже если вы занимаетесь компьютерной музыкой, знания программирования повысят ваш уровень. Сейчас создавать свои программы стало уже не так сложно, как раньше. С помощью таких языков, как C#, и благодаря платформе .NET можно создавать простые утилиты за очень короткое время, и при этом вы не будете ни в чем ограничены. Так что не поленитесь и изучите программирование.

  На протяжении всей книги я буду рассказывать о том, что необходимо знать программисту-хакеру, и покажу множество интересных приемов и примеров на языке C++. Если вы еще плохо знакомы с этим языком, то книга поможет познакомиться с программированием с помощью интересных и познавательных примеров.

- Не тормозить прогресс. Хакеры всегда боролись за свободу информации. Если вы хотите быть хакером, то тоже должны помогать другим. Хакеры обязаны способствовать прогрессу. Некоторые делают это через написание программ с открытым кодом, а кто-то просто делится своими знаниями.

  Открытость информации не означает, что вы не можете зарабатывать деньги. Это никогда не возбранялось, потому что хакеры тоже люди. Самое главное — это созидание. Вот тут проявляется еще одно отличие хакеров от крэкеров: хакеры создают, а крэкеры уничтожают. Если вы написали какую-нибудь уникальную шуточную программу, то это вас делает хакером. Но если вы написали вирус, который уничтожает диск, то это вас делает крэкером, я бы даже сказал — преступником.

  В борьбе за свободу информации может применяться даже взлом, но только не в разрушительных целях. Вы можете взломать какую-нибудь программу, чтобы посмотреть, как она работает, но не убирать ее систем защиты. Нужно уважать других программистов, не нарушать их авторские права, потому что защита программ — это их хлеб.

  Представьте себе ситуацию, если бы вы украли телевизор. Это было бы воровство и преследовалось бы по закону. Многие люди это понимают и не идут на преступления из-за боязни наказания. Почему же тогда крэкеры спокойно ломают программы, не боясь закона? Ведь это тоже воровство. Лично я приравниваю взлом программы к воровству телевизора с полки магазина и считаю это таким же правонарушением.

  Я также отказался от нелегального софта и уже много раз делился размышлениями на эту тему в своем блоге **http://www.flenov.info**.

  Я сам программист и продаю свои программы. Но я никогда не делаю сложных систем защиты, потому что это мешает законопослушным пользователям, а крэкеры все равно взломают. Какие только системы защиты ни придумывали крупные корпорации, чтобы защитить свою собственность, но большинство взламывалось еще до официального выхода программного продукта на рынок. В цивилизованном мире программа должна иметь только простое поле для ввода какого-то кода, подтверждающего оплату, и ничего больше. Не должно быть никаких активаций и сложных регистраций. Но и пользователи должны быть честными, потому что любой труд должен оплачиваться. А то, что какой-то товар (программный продукт) можно получить бесплатно, не значит, что вы должны это делать.
- Не изобретать велосипед. Тут опять действует созидательная функция хакеров. Они не должны стоять на месте и обязаны делиться своими знаниями. Например, если вы написали какой-то уникальный код, то поделитесь им с другими, чтобы людям не пришлось создавать то же самое. Вы можете не выдавать все секреты, но должны помогать другим.

Ну а если к вам попал код другого человека, то не стесняйтесь его использовать (с его согласия!). Не выдумывайте то, что уже сделано другими и обкатано пользователями. Если каждый будет создавать колесо, то никто и никогда не создаст повозку.

- Хакеры — не просто отдельные личности, а целая культура. Но это не значит, что все хакеры одеваются одинаково и все на одно лицо. Каждый из них — это отдельный индивидуум и не похож на других. Не надо копировать другого человека. Удачное копирование не сделает вас продвинутым хакером. Только ваша индивидуальность может сделать вам имя.

  Если вас знают в каких-то кругах, то это считается очень почетным. Хакеры — это люди, добывающие себе славу своими знаниями и добрыми делами. Поэтому любого хакера должны знать.

  Как вам узнать, являетесь ли вы хакером? Очень просто: если о вас говорят как о хакере, то вы и есть хакер. Жаль, что этого добиться очень сложно, потому что большинство считает хакерами взломщиков. Поэтому, чтобы о вас заговорили как о хакере, нужно что-то взломать. Но это неправильно, и не надо поддаваться этому соблазну. Старайтесь держать себя в рамках и добиться славы только добрыми делами. Это намного сложнее, но что поделаешь. Никто не говорил, что будет просто.

- Чем отличаются друг от друга программист, пользователь и хакер? Программист, когда пишет программу, видит, какой она должна быть, и делает так, как он видит. Пользователь не всегда знает, что задумал программист, и использует программу так, как понимает.

  Программист не всегда может предугадать действия своих клиентов, да и программы не всегда тщательно оттестированы. Пользователи имеют возможность ввести параметры, которые приводят к неустойчивой работе программ.

  Хакеры намеренно ищут в программе лазейки, чтобы заставить ее работать неправильно или необычно. Для этого требуется воображение и нестандартное мышление. Вы должны чувствовать исполняемый код и видеть то, чего не видят другие.

  Если вы хотите быть хакером, то должны уметь сделать из абсолютно безобидной вещи что-то интересное и веселое. Для этого вы должны включать свое воображение. Именно оно позволит нам создавать шуточные программы и писать эффективные алгоритмы.

Некоторые считают, что правильно надо произносить "хэкер", а не "хакер". Это так, но только для английского языка. У нас в стране это слово обрусело и стало "хакером". Мы — русские люди, и давайте будем любить свой язык и признавать его правила.

Напоследок советую почитать статью "Как стать хакером". Эта статья написана знаменитым хакером по имени Eric S. Raymond и отражает дух хакерства. Я бы дал ссылку в Интернете, но боюсь, что она может измениться к моменту выхода книги, как это произошло с первым изданием. Поэтому я понадеюсь, что вы умеете пользоваться поисковой машиной (например, yandex.ru) и сможете найти статью самостоятельно.

Тут же возникает вопрос, почему же автор относит к хакерскому искусству написание шуточных и сетевых программ. Попробую ответить на этот вопрос. Во-первых, хакеры всегда пытались доказать свою силу и знания методом написания каких-либо интересных, веселых программ. К этой категории я не отношу вирусы, потому что они несут в себе разрушение, хотя они тоже бывают с изюминкой и юмором. Зато простые и безобидные шутки всегда ценились в узких кругах. Таким способом хакер показывает не только свои знания особенностей операционной системы, но и старается заставить улыбнуться. Не секрет, что многие хакеры обладают хорошим чувством юмора, и он поневоле ищет своего воплощения. Я советую шутить с помощью безобидных программ.

Ну, а сетевое программирование неразделимо с понятием "хакер" с самого его рождения. Хакеры получили распространение благодаря сети, пониманию ее функционирования и большому вкладу в развитие Интернета. Именно поэтому данная тема будет рассматриваться достаточно подробно и, как всегда, с нестандартной точки зрения.

И последнее. Я уже сказал, что любой хакер должен уметь писать программы на каком-нибудь языке программирования. Некоторые заведомо считают, что если человек хакер, то он должен знать и уметь программировать на языке ассемблера. Это не так. Знание этого языка желательно, но не обязательно. Я люблю Delphi, который позволяет мне сделать все, что я захочу. А главное, что я могу сделать это быстро и качественно. Но я также использую C++ и ассемблер.

Языки программирования надо использовать с умом. Несмотря на мою любовь к Delphi должен признать, что он удобен для написания практически любых программ, кроме игр. Тут всегда властвовал и будет властвовать C++. При сетевом программировании иногда может оказаться удобнее C++, а иногда Delphi. Но писать большие приложения на языке ассемблера даже глупо.

Вы также должны понимать необходимость использования технологий. Тут же приведу простейший пример. Сейчас все программисты вставляют в свои продукты поддержку XML. А ведь не всем пользователям этот формат нужен, и не во всех программах он востребован. Следование рекомендациям Microsoft не означает правильность действий, потому что заказчик — не Билл Гейтс, а ваш потребитель. Поэтому надо всегда делать то, что требуется конечному пользователю.

Я рекомендую не обращать особого внимания на авторитет корпорации Microsoft (хотя некоторые разработки гениальны). Сколько технологий доступа к данным придумала Microsoft? Просто диву даешься: DAO, RDO, ODBC, ADO, ADO.NET, и это еще не полный список. Корпорация Microsoft регулярно выкидывает на рынок что-то новое, но при этом сама этим не пользуется. При появлении новой технологии все программисты бросаются переделывать свои программы под новый стандарт и в результате тратят громадные ресурсы на постоянные переделки. Таким образом, конкуренты сильно отстают, а Microsoft движется вперед, потому что не следует своим собственным рекомендациям и ничего не переделывает. Если программа при создании использовала для доступа к данным DAO, то можно спокойно оставить ее работать через DAO и не переделывать на ADO, потому что пользователю все равно, каким образом программа получает данные из базы, главное, чтобы данные были.

Могу привести более яркий пример — интерфейс. В программе MS Office постоянно меняется интерфейс, и при этом всем говорят, что именно он самый удобный для пользователя. Все бегут переводить свои программы на новый внешний вид меню и панелей, а тот же Internet Explorer и все остальные программы выглядят как 10 лет назад. В них практически ничего не меняется, и Microsoft не тратит на это время, а конкуренты тратят месяцы на переписывание множества строчек кода.

Да, следование моде придает вашим программам эффектность, но при этом вы должны сохранить индивидуальность.

Возможно, сложилось впечатление, что я противник Microsoft, но это не так. Мне очень нравятся продукты этой фирмы, например, Windows или MS SQL Server, но я не всегда согласен с ее методами борьбы с конкурентами. Это жестокий бизнес, но не до такой же степени!

Программисты и хакеры навязывают другим свое мнение о любимом языке программирования как единственно приемлемом, обычно добиваясь успеха, ведь заказчик часто не понимает в программировании. На самом же деле заказчику все равно, на каком языке вы напишете программу, его интересуют только сроки и качество. Лично я могу обеспечить минимальные сроки написания приложения, сохраняя хорошее качество, только работая на Delphi. Такое же качество на C++ я (да и любой другой программист) смогу обеспечить только в значительно большие сроки.

В данной книге будут описываться примеры именно на Visual C++, потому что этот язык является наиболее распространенным и многими признан стандартом для программирования. Однако существует книга, в которой решаются те же проблемы, но с использованием примеров на языке Delphi.

Вот когда заказчик требует минимальный размер или наивысшую скорость работы программы, тогда я берусь за ассемблер и С (не путать С и С++). Но это бывает очень редко, потому что сейчас носители информации уже практически не испытывают недостатка в размерах, и компьютеры работают в миллионы раз быстрее своих предков. Таким образом, размер и скорость программы уже не являются критичными, и на первый план ставятся скорость и качество выполнения заказа.

Итак, на такой деловой ноте мы закончим вводную лекцию и перейдем к практическим упражнениям по воинскому искусству, где часто главное — скрытность и победа минимальными силами.

Благодарности

Первым делом хочу поблагодарить своих родителей за то, что они произвели меня на свет. Если бы ни они, не было бы этой книги, по крайней мере, в моем исполнении. Свою жену Ирину, которая поддерживала меня. Свою дочку Юлю и сына Кирилла, которые иногда давали мне время для работы, а иногда отнимали его, из-за чего случалось отдыхать от компьютера.

Редакторов журналов, где публиковались мои статьи. Благодаря этому я получил большой опыт журналистики, который впоследствии помог мне написать мои книги. Я технический специалист, и еще некоторое время назад не мог связать двух слов, не говоря уже о том, чтобы красиво и доступно изложить свои мысли на компьютере. Но благодаря большой журналистской практике в журнале "Хакер" я немного научился писать.

Отдельное спасибо издательству "БХВ-Петербург" за то, что поверили в меня как в автора и помогли в издании моих книг. Благодарю всех редакторов издательства за то, что редактируют мой бред и исправляют грамматические ошибки, которых иногда бывает очень много.

Дальше я могу еще очень долго перечислять людей, которым благодарен в своей жизни, поэтому коротко: воспитателям в детсаде, учителям в школе, преподавателям в институте, всей команде журнала "Хакер" и всем моим друзьям за то, что они меня терпят, вероятно, любят, и по возможности помогают. Ну, и самое главное, я хочу поблагодарить всех моих читателей, потому что вдохновляют на новые свершения.

А вам отдельное спасибо за то, что приобрели эту книгу, отдельное спасибо, если именно приобрели, а не скачали в Интернете. Надеюсь, что вы не разочаруетесь и не будете жалеть о потраченных деньгах и времени. Чтобы этого не случилось, я постарался наполнить информацией не только страницы книги, но и компакт-диск, где находится много дополнительной и полезной информации.

ГЛАВА 1



Оптимизация

В этой главе мы будем говорить об оптимизации, причем — с неожиданных сторон. Мы будем оптимизировать видимость окон, мы будем уменьшать размер программы, повышать скорость работы кода и даже защищать его. Какое отношение имеет защита к оптимизации? Иногда эти темы являются антиподами, и именно поэтому я решил объединить эти две темы под одной крышей, а точнее — в одной главе.

Что самое главное при написании программ-приколов? Ну, конечно же, невидимость. Программы, созданные в этой и следующих главах, будут незаметно сидеть в системе и выполнять нужные действия при наступлении определенного события. Это значит, что программа не должна отображаться на Панели задач или в списке запущенных программ, в окне, появляющемся при нажатии <Ctrl>+<Alt>+<Del>. Поэтому, прежде чем начать что-то писать, нужно узнать, как спрятать свое творение от чужого глаза.

Кроме этого, программы-приколы должны иметь маленький размер. Приложения, создаваемые Visual C++ с использованием современных технологий MFC (Microsoft Foundation Classes, базовые классы от Microsoft), достаточно "весомые". Даже простейшая программа, выводящая одно окно, займет достаточно много места на диске. Если вы захотите отослать такую шутку по электронной почте, то отправка и получение письма с вашей программой отнимут лишнее время у вас и получателя. Это не очень приятно, поэтому в этой главе мы познакомимся с тем, как можно уменьшить размер программ, создаваемых в Visual C++.

1.1. Сжатие исполняемых файлов

Самый простой способ уменьшить размер приложения — использование программы для сжатия файлов. Раньше я любил использовать ASPack (**http:/**

www.aspack.com), но в последнее время перешел на использование открытой библиотеки UPX. Эта библиотека распространяется в исходных кодах и может быть найдена по адресу **http://upx.sourceforge.net**. Для работы с библиотекой лучше иметь какую-нибудь визуальную программу. Тут я предпочитаю разработку Абдульманова Рафаэля Рахимовича (**http://rafsoft.narod.ru**) — UPX Control. В принципе, программа простая, но удобная. Главное окно программы можно увидеть на рис. 1.1.

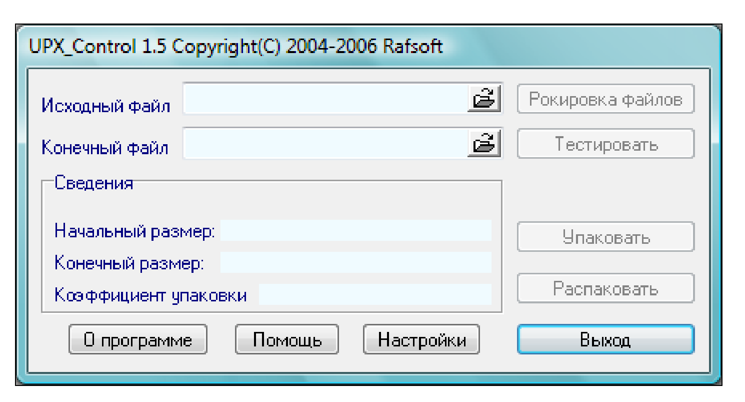


Рис. 1.1. Главное окно утилиты управления UPX

Окно простейшее, ибо нужно только выбрать исполняемый файл, который вы хотите сжать, результирующий файл — и нажать кнопку **Упаковать**. Этой же программой можно вернуться к изначальному состоянию, нажав кнопку **Распаковать**.

Теперь давайте разберемся, как работает сжатие. Сначала весь код программы сжимается архиватором. В программе используется алгоритм сжатия LZWA, оптимизированный для сжатия двоичного кода. В конец сжатого кода добавляется код разархиватора. И в самом конце ASPack изменяет заголовок исполняемого файла так, чтобы при старте сначала запускался разархиватор, который будет во время выполнения распаковывать программу в первоначальное состояние прямо в памяти и запускать распакованный вариант на исполнение.

В UPX алгоритм сжатия очень хороший, а разархиватор достаточно маленький (меньше 1 Кбайт), поэтому сжатие происходит очень сильно, а к результирующему файлу добавляется только один килобайт. Таким образом, программа может сжать файл размера в 1,5 Мбайт в 300–500 Кбайт.

Теперь, когда вы запускаете сжатую программу, сначала заработает разархиватор, который разожмет бинарный код программы и аккуратно поместит его в памяти компьютера. Как только этот процесс закончится, разархиватор передаст управление вашей программе.

Некоторые считают, что из-за расходов на распаковку программа будет работать медленней!!! Я бы сказал, что вы не заметите разницу. Даже если и будут какие-то потери, то они будут неощутимы (по крайней мере, на современных компьютерах). Это происходит потому, что архивация хорошо оптимизирована под двоичный код. И по сути дела, распаковка происходит только один раз и в дальнейшем никакого влияния на работу программы не оказывает. В результате, потери в скорости из-за сжатия будут неощутимы.

Программа без сжатия перед началом выполнения все равно грузится в память. В случае сжатого кода во время загрузки происходит разархивирование кода. В данном способе есть две стороны: происходят затраты времени на распаковку, но программа меньше занимает места на диске и быстрее считывается с него. Жесткий диск — одно из самых медленных звеньев персонального компьютера, поэтому чем меньше надо загружать, тем быстрее программа может приступить к выполнению. Именно вследствие этого итоговая потеря в скорости запуска программы незначительная.

При нормальном программировании с использованием всех современных возможностей типа визуальности и объектного программирования код получается большим, но его можно сжать специальным архиватором на 60–70%. А писать такой код намного легче и быстрее.

Еще одно "за" использование сжатия — заархивированный код труднее взломать, потому что не каждый дизассемблер сможет прочитать упакованные команды. Так что помимо уменьшения размера вы получаете защиту, способную остановить большинство взломщиков. Конечно же, профессионала этим не отпугнешь. Но взломщик средней руки не будет возиться со сжатым двоичным кодом.

Тут нужно быть честным и вспомнить, что и описанная ранее утилита работы с UPX умеет разархивировать, а значит, хакер тоже сможет ей воспользоваться. Декомпиляторы и отладчики тоже не из каменного века, и хорошие разработки умеют определять алгоритм сжатия и распаковывать код автоматически при открытии файла без дополнительных программ.

Коммерческие программы сжатия, такие как ASPack, имеют функции шифрования, которые могут усложнить жизнь хакерам. Опять же, только усложнить, но не испортить вовсе. Для исполнения код в любом случае будет расшифрован, а в арсенале взломщиков есть утилиты, позволяющие снимать код из памяти для дальнейшего анализа. Этот процесс не из легких, но вполне возможный.

1.2. Без окон, без дверей...

Следующий способ уменьшить размер программы заключается в ответе на вопрос, из-за чего программа, созданная в Visual C++, получается большой.

Ответ очень прост: C++ является объектно ориентированным языком. В нем каждый элемент представляет собой объект, который обладает своими свойствами, методами и событиями. Любой объект вполне автономен и многое умеет делать без ваших указаний. Это значит, что вам нужно только подключить его к своей форме, изменить нужным образом свойства — и приложение готово! И оно будет работать без какого-либо прописывания его деятельности.

Но в объектном программировании есть и свои недостатки. В объектах реализовано большое количество действий, которые вы и пользователь сможете производить с ними. Но реально в любой программе мы используем два-три из всех них. Все остальное — для программы лишний груз, который никому не нужен.

Но как же тогда создать компактный код, чтобы программа занимала минимум места на жестком диске и в оперативной памяти? Тут есть несколько вариантов:

- не использовать библиотеку MFC, которая упрощает программирование на языке C++. В этом случае придется больше писать кода вручную и работать только с Windows API (Windows Application Programming Interface, прикладной программный интерфейс). Программа получается очень маленькой и быстрой. Результирующий код будет меньше, чем при использовании MFC в сочетании с самым большим сжатием. Но таким образом вы лишаетесь простоты визуального программирования и можете ощутить все неудобства программирования с помощью чистого WinAPI.

  Для большей оптимизации размера файла можно использовать ассемблер, но он относительно сложен, да и писать программу на нем намного дольше, чем даже на чистом C. Хотя кому как. Именно поэтому данная тема не рассматривается в этой книге;
- сжимать готовые программы с помощью компрессоров. Объектный код сжимается в несколько раз, и программа, созданная с использованием MFC, может превратиться из монстра в 300 Кбайт в скромного по размерам "зверя", занимающего на диске всего 30–50 Кбайт. Главное преимущество состоит в том, что вы не лишаетесь возможностей объектного программирования и можете спокойно забыть про неудобства WinAPI.

Второй метод мы уже обсудили *(см. разд. 1.1)*, поэтому остается только описать первый, и самый интересный, вариант. Полученный код уже получается маленьким, и его можно еще дополнительно сжать с помощью утилиты UPX. Результат будет потрясающим.

Если вы хотите создать действительно компактную программу, то необходимо забыть про все удобства. Вы не сможете подключать визуальные формы или другие удобные модули, написанные фирмой Microsoft для упрощения

жизни программиста. Нельзя использовать классы или компоненты ActiveX. Только функции API самой ОС Windows — и ничего больше.

Теперь переходим к практическим занятиям. Запустите Visual C++ и создайте новый проект. Для этого нужно выбрать команду меню **File | New | Project** (Файл | Новый | Проект). Перед вами откроется окно создания нового проекта (рис. 1.2). Слева расположено дерево типов проектов. Нас интересует C++, поэтому выберите пункт **Visual C++**. Этот пункт мы будем выбирать при написании абсолютно всех примеров из данной книги. С правой стороны в списке **Templates** (Шаблоны) варианты создания различных проектов. Выберите пункт **MFC Application** (Приложение MFC).

Внизу окна расположены две строки ввода. В первой вы должны указать имя создаваемого проекта. Оно будет использоваться в качестве имени запускаемого файла и имени файла, который вы будете в дальнейшем открывать для редактирования. Давайте здесь укажем: `TestMFC`.

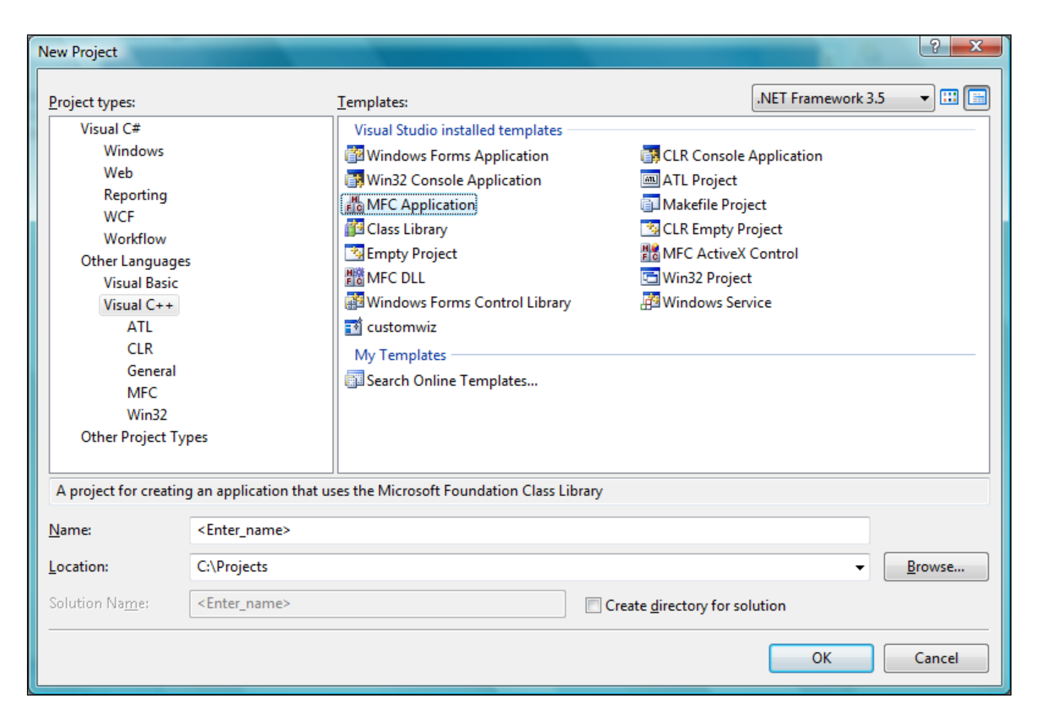


Рис. 1.2. Окно выбора типа проекта

В строке **Location** (Расположение) нужно указать путь к папке, в которой среда разработки создаст необходимые проекту файлы. Я рекомендую завести свою папку с именем, например, My C++ Projects (или любым другим на

выбор, но понятным по смыслу), где будут размещаться все проекты. Выберите эту папку и нажмите **OK**. По окончании работы мастера у вас в папке My C++ Projects появится еще одна папка с именем TestMFC, в которой и будут находиться все файлы данного проекта.

Как только вы нажали **OK** в окне создания нового проекта (см. рис. 1.2), перед вами откроется окно Мастера создания нового MFC-приложения (рис. 1.3).

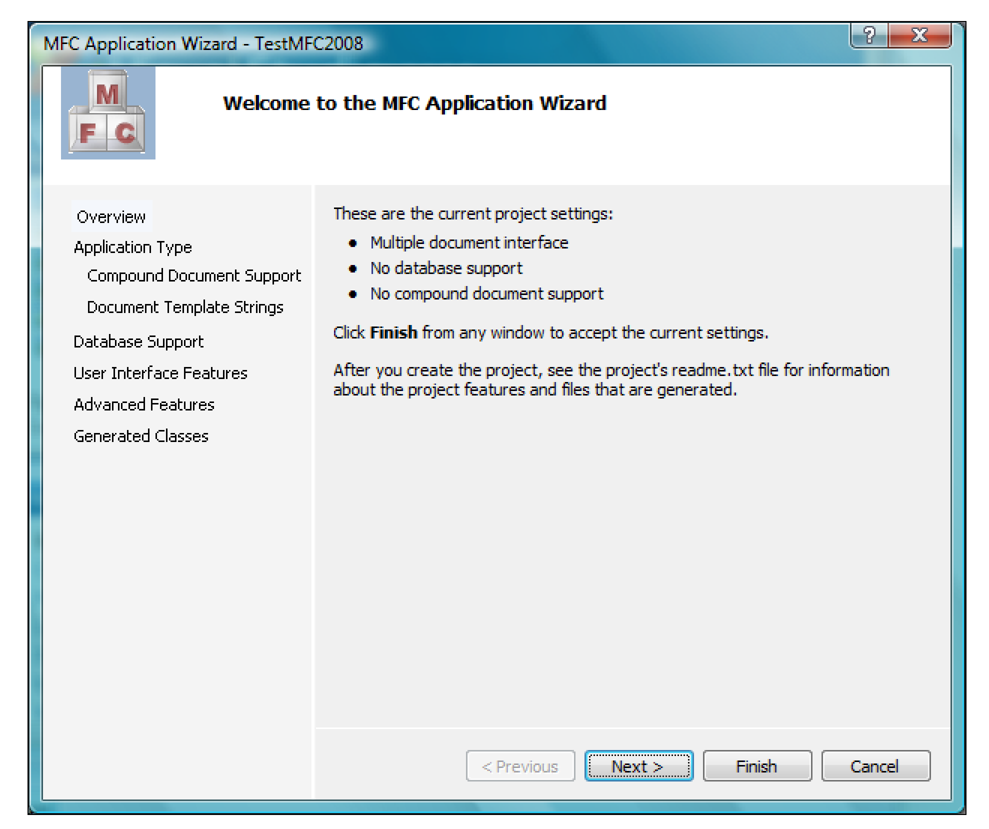


Рис. 1.3. Окно Мастера создания проекта

Вы можете сразу нажать кнопку **Finish**, чтобы завершить его работу с параметрами по умолчанию, или предварительно указать свои настройки. Наша задача — создать маленькое приложение, поэтому на данном этапе постараемся оптимизировать то, что может создать для нас Мастер.

С левой стороны окна расположены разделы. Выделяя их, вы можете настраивать соответствующие параметры. Давайте просмотрим все разделы и установим необходимые значения, а заодно познакомимся с теми параметра-

ми, которые будут использоваться для создания приложений при рассмотрении последующих примеров:

- **Application Type** (Тип приложения) — в этом разделе мы задаем тип создаваемого приложения. Давайте укажем здесь следующие значения:
  - **Single Document** (Документ с одним окном) — нам достаточно будет только одного окна. Многодокументные приложения мы не рассматриваем, поэтому большинство примеров с использованием MFC будет основываться на этом типе приложений или на основе диалоговых окон (**Dialog based**);
  - **Project style** (Стиль проекта) — во всех приложениях будем использовать стиль по умолчанию (**MFC стандарт**);
  - **Document | View architecture support** (Поддержка архитектуры Документ | Просмотр) — это значение нас пока не интересует, поэтому оставим по умолчанию;
- **Advanced Features** (Дополнительные возможности) — в этом разделе в будущем нас будет интересовать только параметр **Windows socket** (поддержка сокетов Windows), который позволит нам писать примеры для работы с сетью.

Во всех остальных разделах оставляем значения по умолчанию, потому что мы не будем использовать базы данных или документы. В большинстве случаев нам будет достаточно окна и меню. А первое время постараемся обходиться вообще без MFC.

Нажмите кнопку **Finish**, чтобы завершить работу мастера. По завершении его работы вы увидите главное окно среды разработки Microsoft Visual C++, представленное на рис. 1.4. Это окно мы будем использовать достаточно часто, поэтому в процессе работы уточним все детали.

Сейчас нас интересуют параметры проекта. Мы должны отключить все, что нам будет мешать. При сборке проекта в Visual C++ по умолчанию могут использоваться два вида настроек: debug и release. Первый необходим на этапе разработки, и в этом режиме Visual C++ создает запускаемый файл, который содержит слишком много дополнительной информации. Она будет необходима вам в среде разработки в дальнейшем при отладке программы и поиске ошибок. Во втором режиме эта информация отключается, и запускаемый файл будет меньшего размера.

В верхней части окна на панели с кнопками найдите выпадающий список, в котором написано **Debug**. Измените это значение на **Release**.

Среда разработки Visual C++ может создавать запускаемые файлы, использующие MFC-библиотеки двух типов: статические и динамические. По умолчанию используется динамическая сборка. В таком режиме запускаемый

файл получается меньшего размера, но он не будет работать без динамических библиотек, таких как mfcXXX.dll, где XXX — это номер версии среды разработки.

Рис. 1.4. Окно среды разработки

В этом случае, чтобы кто-то смог запустить наш проект, мы должны отослать ему не только запускаемый файл, но и библиотеки. Это неудобно и неприлично. Лучше использовать статическую компиляцию, при которой результирующий файл будет намного больше, зато все будет содержать внутри себя. При таком подходе не потребуются дополнительные библиотеки.

Чтобы изменить тип использования MFC, в окне **Solution Explorer** сначала выберите имя вашего проекта, а затем в меню команду **Project/Properties**. На рис. 1.5 представлено окно свойств проекта.



После выхода первой книги я получил много вопросов, когда пользователи не смогли скомпилировать проекты. Это случилось у тех, у кого по умолчанию в IDE выбирается Multibyte-строки. Я не использовал Unicode-строки, поэтому если компилятор будет ругаться, то в параметре **Character Set** выберите значение **No Set**.

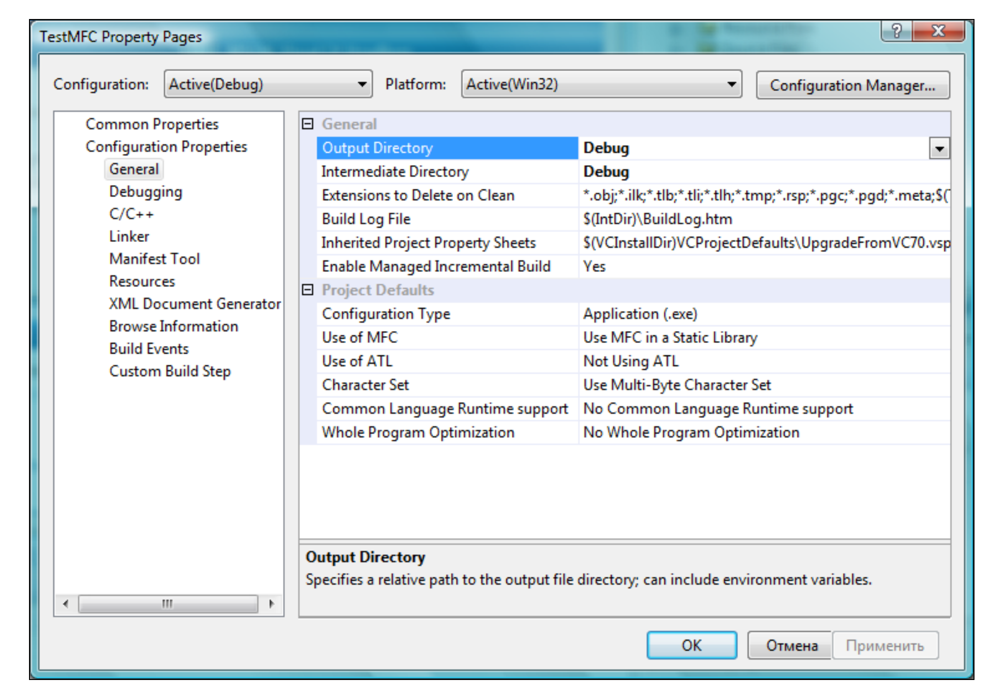


Рис. 1.5. Окно свойств проекта

Слева в окне расположены разделы свойств. Нас будет интересовать раздел **General** (Основные). Выделите его, и в основном окне появится список соответствующих свойств. Найдите свойство **Use of MFC** и измените его значение на **Use MFC in a Static Library**. Нажмите кнопку **OK**, чтобы закрыть окно и сохранить изменения.

Теперь соберем наш проект в готовый исполняемый файл. Для этого нужно выбрать команду меню **Build | Build solution** (Построить | Построить проект). Внизу главного окна, в панели **Output** (Вывод), будет появляться информация о ходе сборки. Дождитесь, пока не появится сообщение типа:

```
---------------------- Done ----------------------
    Build: 1 succeeded, 0 failed, 0 skipped
```

Теперь перейдите в папку, которую вы выделили под хранение проектов, и найдите там папку TestMFC. В ней расположены файлы с исходным кодом нашего проекта, сгенерированные мастером. Тут же должна быть папка Release, в которой среда разработки создала во время компиляции промежуточные и исполняемый файлы. Выделите файл TestMFC.exe и посмотрите его свойства (надо щелкнуть правой кнопкой мыши и выбрать в появившемся меню пункт **Свойства**). Размер нашего пустого проекта у меня в Visual

Studio 2008 составил 446 Кбайт. От такого размера не только звезда будет в шоке, но и простые программисты, как мы! Это очень много. В Visual Studio 2003 это было около 380 Кбайт.

Попробуйте открыть его в программе UPX и сжать. У меня сжатый исполняемый файл составил 209 Кбайт. Сжатие составило более 50%, и это уже более или менее приемлемый размер для шуточной программы.



Пример этой программы вы можете увидеть на компакт-диске в папке /Demo/Chapter1/TestMFC. Чтобы открыть этот пример, выберите команду меню **File | Open solution**. Перед вами появится стандартное окно открытия файлов. Перейдите в нужную папку и выберите файл с именем проекта и расширением .vcproj или .sln.

Чтобы сделать программу еще меньше, необходимо отказаться от MFC и писать на чистом C. Это немного сложнее и не так удобно, но для небольших проектов вполне приемлемо.

Для того чтобы создать маленькую программу без использования MFC, нужно снова использовать меню **File | New | Project** и здесь выбрать уже тип создаваемого проекта **Win32 Project**. В качестве имени давайте укажем `ctest`, а путь оставим тот же.

Если у вас все еще открыт предыдущий проект, то под строкой ввода пути для проекта есть переключатели: **Add to solution** (Добавить в решение) и **Close solution** (Закрыть решение). Если вы выберете первый из них, то текущий проект будет добавлен в уже открытый. Если выбрать закрытие, то текущий проект будет закрыт, и для вас будет создано новое рабочее поле.

После нажатия кнопки **OK** перед вами откроется окно Мастера. Первый шаг чисто информационный, поэтому выберите раздел **Application Settings** (Настройки приложения). Перед вами откроется окно, как на рис. 1.6.

Нас интересует простое приложение Windows, поэтому вы должны выбрать в разделе **Application type** (Тип приложения) переключатель **Windows application**. Больше ничего, чтобы мастер не добавлял ничего лишнего. Нам необходим только самый минимум. Нажмите кнопку **Finish**, и будет создан новый проект.

Здесь также нужно изменить **Debug** на **Release**, чтобы создать проект без дополнительной информации. В настройках проекта ничего менять не надо, потому что созданный мастером шаблон не использует MFC, и ему не нужны динамические библиотеки. Можете зайти в свойства проекта и убедиться, что в свойстве **Use of MFC** стоит **Standard Windows Libraries** (Использовать стандартные библиотеки Windows). Это значит, что нет MFC, и ничего до-

полнительного программе не надо, только стандартные библиотеки Windows и прямой вызов API операционной системы.

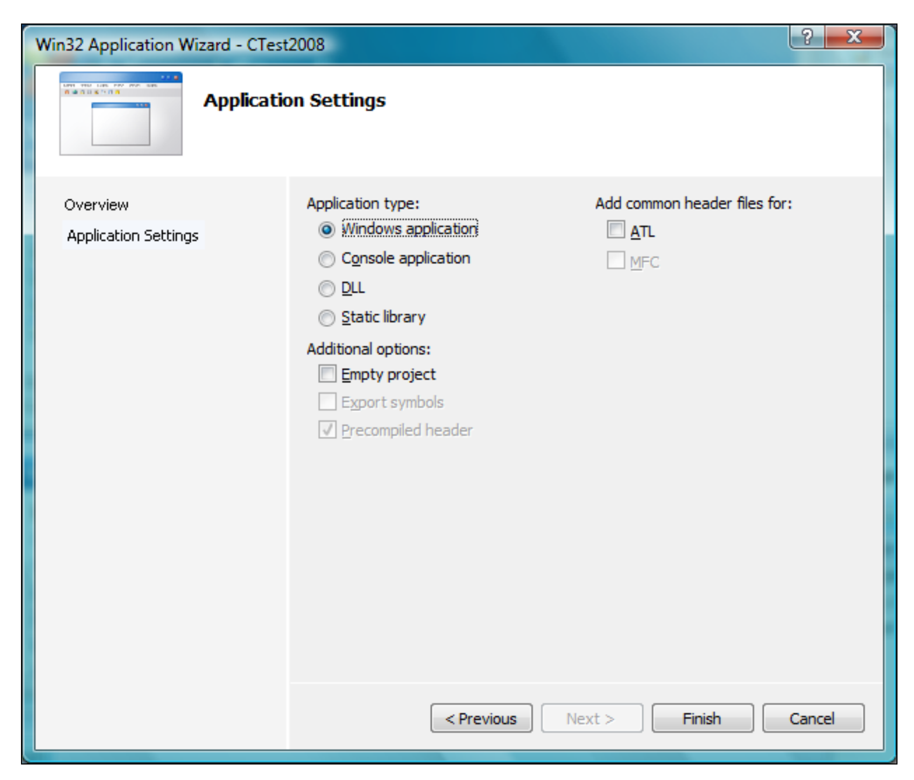


Рис. 1.6. Параметры приложения Win32

Попробуйте откомпилировать проект. Для этого выберите команду меню **Build | Build solution**. По окончании сборки перейдите в папку ctest/Release каталога, где вы создавали проекты, и посмотрите на размер результирующего файла. У меня в Visual Studio 2008 получилось 86 Кбайт. В Visual Studio 2003 размер составлял 81 Кбайт. Если сжать файл, то получится менее 70 Кбайт. Вот такая программа уже скопируется по сети достаточно быстро.

Программу можно еще больше оптимизировать и убрать некоторые вещи, которые не используются, но отнимают драгоценные килобайты, однако мы этого делать уже не будем. Сделаем только пару выводов: размер файла зависит от используемых функций и может отличаться в зависимости от используемого компилятора. Каждый компилятор может использовать разные функции и методы оптимизации.

Если вы не имеете опыта программирования, то я бы порекомендовал сейчас отложить книгу и прочитать GetStarted.doc на компакт-диске в каталоге Doc и

содержимое Doc\For Beginer, где я выложил дополнительную информацию для начинающих программистов, чтобы материал книги был понятен всем и интересен программистам с разным уровнем подготовки.

1.3. Оптимизация программ

Вся наша жизнь — это борьба с тормозами и нехваткой времени. Каждый день мы тратим по нескольку часов на оптимизацию. Каждый из нас старается оптимизировать все, что попадает под руку. А вы уверены, что вы это делаете правильно? Может быть, есть возможность что-то сделать еще лучше?

Я понимаю, что все сейчас разленились и выполняют свои обязанности спустя рукава. Лично я до такой степени привык, что за меня все делает компьютер, что даже забыл, как выглядит шариковая ручка. Недавно мне довелось писать заявление на отпуск на простой бумаге, так я долго вспоминал, как пишется буква "ю". Я помню, что эта клавиша находится возле правой клавиши <Shift>, но как ее писать... Пришлось подглядывать, как она выглядит на клавиатуре. Это не шутка. Это прогресс, благодаря которому я все делаю на компьютере.

Даже для того, чтобы написать текст из двух строк, мы включаем свой компьютер и загружаем MS Word, тратя на это драгоценное время. А может, легче было бы написать этот текст на бумаге?

Программисты считают, что раз их творение (в виде исходного кода) никто не увидит, то можно писать что угодно. Так это они ошибаются. С этой точки зрения программы с открытым исходным кодом имеют большое преимущество, потому что намного чище и быстрей. Создавая код, мы ленимся его оптимизировать не только в плане размера, но и в плане скорости. Глядя на такие вещи, хочется ругаться непристойными словами, только программа от этого лучше не станет.

Хакеры далеко не ушли. Если раньше, глядя на программиста или хакера, возникал образ прокуренного, заросшего и немытого молодого человека, то сейчас это цифровое существо, за которое все выполняют машины.

Все это деградация! Мы берем в руки мышку и начинаем тыкать ею, где попало, забывая про клавиатуру и горячие клавиши. Мышка была придумана в качестве помощника, а не заменителя клавиатуры. Я считаю, что надо бороться с этим. Меня самого некоторое время назад иногда разбирала такая лень, что я убирал клавиатуру, запускал экранную клавиатуру и начинал работать только мышкой. Осталось только покрыть мое тело шерстью и посадить в клетку к таким же ленивым шимпанзе.

Но теперь я работаю с ноутбуком, его клавиатуру не убрать. Работать с Touch Pad не так удобно, поэтому про клавиатуру никак не забыть.

Не надо тратить большие деньги на модернизацию компьютера. Лучше начните изменения с себя. Давайте оптимизируем свою работу и свои творения, и тогда компьютер заработает намного быстрее.

Изначально эта часть книги задумывалась как рассказ об оптимизации кода программ. Но впоследствии объединил советы с реальной жизнью, потому что оптимизировать надо все. Я буду говорить про теорию оптимизации, а ее законы действуют везде. По одним и тем же законам вы можете оптимизировать свой распорядок дня, чтобы успевать все сделать, и свою ОС, чтобы она работала быстрей. Но основное все же будет относиться к коду программ.

Как всегда, я постараюсь давать больше реальных примеров, чтобы вы могли применить все сказанное на практике.

Начну я с законов, которые работают не только в программировании, но и в реальной жизни. Ну, а напоследок оставлю только то, что может пригодиться при оптимизации кода.

Закон № 1

Оптимизировать можно все. Даже там, где вам кажется, что все и так работает быстро, можно сделать еще быстрее.

Это действительно так. И этот закон очень сильно проявляется в программировании. Идеального кода не существует. Даже простую операцию сложения 2 + 2 тоже можно оптимизировать. Чтобы достичь максимального результата, нужно действовать последовательно и, желательно, в том порядке, в котором описано далее.

Помните, что любую задачу можно решить хотя бы двумя способами (если не больше), и ваша задача — выбрать наилучший метод, который обеспечит желаемую производительность и универсальность.

Закон № 2

Первое, с чего нужно начинать, — это поиск самых слабых и медленных мест.

Зачем начинать оптимизацию с того, что и так работает достаточно быстро! Если вы будете оптимизировать сильные места, то можете встретить неожиданные конфликты. Да и эффект будет минимален.

Тут же я вспоминаю пример из своей собственной жизни. Примерно в 1996 году меня посетила одна невероятная идея — написать собственную игру в стиле Doom. Я не собирался ее делать коммерческой, а хотел только потренировать мозги на сообразительность. Четыре месяца невероятного труда, и нечто похожее на движок уже было готово. Я создал один голый

уровень, по которому можно было перемещаться, и с чувством гордости побежал по коридорам.

Никаких монстров, дверей и атрибутики в нем не было, а тормоза ощущались достаточно значительные. Тут я представил себе, что будет, если добавить монстров и атрибуты, да еще и наделить все это AI... Вот тут чувство собственного достоинства поникло. Кому нужен движок, который при разрешении 320×200 (тогда это было круто!) в голом виде тормозит со страшной силой? Вот именно...

Понятное дело, что мой виртуальный мир нужно было оптимизировать. Целый месяц я бился над кодом и вылизывал каждый оператор моего движка. Результат — мир стал прорисовываться на 10% быстрей, но тормозить не перестал. И тут я увидел самое слабое место — вывод на экран. Мой движок просчитывал сцены достаточно быстро, а пробоиной был именно вывод изображения. Тогда еще не было шины AGP, и я использовал простую PCI-видеокарту от S3 с 1 Мбайт памяти. Пара часов колдовства, и я выжал из PCI все возможное. Откомпилировав движок, я снова загрузился в свой виртуальный мир. Одно нажатие клавиши "вперед", и я очутился у противоположной стены. Никаких тормозов, сумасшедшая скорость просчета и моментальный вывод на экран.

Как видите, моя ошибка была в том, что вначале я неправильно определил слабое место своего движка. Я месяц потратил на оптимизацию математики, и что в результате? Мизерные 10% прироста производительности. Но когда я реально нашел слабое звено, то смог повысить этот параметр в несколько раз.

Именно поэтому я говорю, что надо начинать оптимизировать только со слабых мест. Если вы ускорите работу самого слабого звена вашей программы, то, может быть, и не понадобится ускорять другие места. Вы можете потратить дни и месяцы на оптимизацию сильных сторон и увеличить производительность только на 10% (что может оказаться недостаточным) или несколько часов на совершенствование слабой части и получить улучшение в 10 раз!

Слабые места компьютера

Меня поражают люди, которые гонятся за мегагерцами процессора и сидят на доисторической видеокарте от S3, жестком диске на 5400 оборотов и с 32 Мбайт памяти. Посмотрите в корпус своего компьютера и оцените его содержимое. Если вы увидели, что памяти у вас не более 256 Мбайт, то встаньте и громко произнесите: "Уважаемый DIMM, команда выбрала вас. Вы сегодня самое слабое звено и должны покинуть мой компьютер". После этого покупаете себе гигабайт (а лучше — 2) памяти и наслаждаетесь ускорением работы Visual C++, Photoshop и других "тяжелых" программ.

В данном случае наращивание сотни мегагерц у процессора даст меньший прирост в скорости. Если вы используете "тяжелые" приложения при нехватке памяти, то процессор начинает тратить слишком много времени на загрузку и выгрузку данных в файл подкачки. Ну а если в вашем компьютере достаточно оперативной памяти, то процессор уже занимается только расчетами и не расходует драгоценное время на лишние загрузки/выгрузки.

То же самое с видеоадаптером. Если видеокарта у вас слабенькая, то процессор будет просчитывать сцены быстрей, чем они будут выводиться на экран, или центральному процессору придется брать на себя расчеты, которые можно было возложить на видеочип. Поэтому наращивание тактовой частоты процессора грозит простоями и минимальным приростом производительности.

Поиск слабых мест

Как правильно определить слабое место программы? Что, если вы неправильно определите слабое место и зря потратите время? Все очень просто, ибо можно воспользоваться специализированными утилитами, которые помогут найти функции, на выполнение которых процессор тратит больше всего времени. Наиболее популярной является разработка компании Intel — VTune Performance Analyzer.

Эта программа анализирует выполнение кода и определяет наиболее медленно выполняемые участки, отображая их графически. Именно на них нужно обратить внимание. Сократив их всего на несколько тактов, можно выиграть драгоценные секунды в выполнении.

Несмотря на то, что VTune Performance Analyzer является отдельным продуктом совершенно самостоятельной компании, он хорошо интегрируется с Visual Studio и поддерживает множество языков программирования: C, C++, Java, C#, Fortran, .NET и др. Поскольку разработчиком программы является создатель популярного процессора, кто, как не он, может знать методы оптимизации для своих процессоров, и кто, как не Intel, может предложить варианты решения!

Закон № 3

Следующим шагом вы должны разобрать по косточкам все операции и выяснить, где они регулярно повторяются. Начинать оптимизацию нужно именно с них.

Опять начнем рассмотрение этого закона с программирования. Допустим, что у вас есть следующий код (приведена просто логика, а не реальная программа):

1. `А:=А*2;`
2. `Б:=1;`
3. `Х:=Х+Б;`
4. `Б:=Б+1;`
5. `Если Б<100, то перейти на шаг 3.`

Любой программист скажет, что здесь слабым местом является первая строка, потому что там используется умножение. Это действительно так. Умножение всегда выполняется дольше, и если заменить его на сложение (`А:=А+А`) или, еще лучше, на сдвиг, то вы выиграете пару тактов процессорного времени. Но это только пару тактов, и для процессора это будет незаметно.

Теперь посмотрите еще раз на наш код. Больше ничего не видите? А я вижу. В этом коде используется цикл: "Пока Б<100, будет выполняться операция Х:=Х+Б". Это значит, что процессору придется выполнить 100 переходов с шага 5 на шаг 3. А это уже немало. Как можно здесь что-то оптимизировать? Очень легко. В этом месте у нас выполняются две строки: 3 и 4. А что, если мы внутри цикла размножим их 2 раза:

1. `Б:=1;`
2. `Х:=Х+Б;`
3. `Б:=Б+1;`
4. `Х:=Х+Б;`
5. `Б:=Б+1;`
6. `Если Б < 100, то перейти на шаг 3.`

Здесь мы видоизменили цикл. Вторую и третью операции мы повторили два раза. Это значит, что за один проход нового цикла выполняются два раза строки 3 и 4, и только после этого произойдет переход на строку 3 для повторения операции. Такой цикл уже нужно повторить только 50 раз (потому что за один проход выполняется два действия). Это значит, что мы сэкономили 50 операций переходов. Неплохо? А это уже несколько сотен тактов процессорного времени.

А что, если внутри цикла написать строки 2 и 3 десять раз. Это значит, что за один проход цикла строки 2 и 3 будут вычисляться 10 раз, и мне понадобится повторить такой цикл только 10 раз, чтобы получить в результате 100. А это уже экономия 90 операций переходов.

Недостаток этого подхода — увеличился код нашей программы, зато повысилась скорость, и очень значительно. Этот способ очень хорош, но им не стоит злоупотреблять. С одной стороны, увеличивается скорость, а с другой — увеличивается размер. А большой размер — это враг любой программы. Поэтому надо находить золотую середину.

В любом деле главное — разумная достаточность. Чем больше вы увеличиваете код ради оптимизации скорости, тем меньше результирующий эффект от оптимизации.

В жизни таких примеров намного больше. Любую циклическую операцию можно оптимизировать. Хотите пример? Пожалуйста. Допустим, у вашего провайдера Интернета есть несколько телефонов доступа. Вы каждый день названиваете на каждый из них в ожидании найти свободный. Начинающий тут же скажет, что провайдер обязан оптимизировать свои пулы модемов в один, чтобы не надо было трезвонить по всем номерам сразу. Но опытный пользователь должен знать, что не у каждого есть хорошая связь с любой телефонной станцией города. Поэтому провайдеры держат пулы на разных станциях, чтобы вы могли выбрать тот, с которым связь лучше. Поставьте программу дозвона (таких сейчас полно в Интернете), которая сама будет перебирать номера телефонов.

А теперь другой пример: вам на 1 час досталась карточка какого-то нового провайдера. Заносить ее в программу дозвона не имеет смысла, потому что вы можете больше никогда не позвонить ему: выигрыш практически нулевой, потому что пока вы меняете настройки, уже можно было дозвониться стандартными средствами Windows. Отсюда сразу же напрашивается вывод — нужно правильно выбирать средства для выполнения необходимых задач.

Закон № 4

Этот закон — расширение предыдущего. Оптимизировать одноразовые операции — это только потеря времени.

Как-то я прочитал рассказ в Интернете "Записки жены программиста" (**http://www.exler.ru/novels/wife.htm**). Очень даже жизненный рассказ. Когда я его читал, у меня было ощущение, что его написала моя жена. Слава "Красной Шапочке", что она на такую подлость не способна. Так вот там была такая ситуация.

Очаровательная девушка выходит замуж за программиста, и им надо разослать приглашения на свадьбу. Вместо того чтобы просто набрать их, программист кричит, что он крутой, и пишет специальную программу. Ее разработка заняла один день, и столько же — отладка.

Главная ошибка — неправильная оптимизация своего труда. Легче подготовить шаблон в любом текстовом редакторе, а потом только менять фамилии приглашенных. Но даже если нет текстового редактора, писать программу действительно нет смысла. Затраты большие, а пользоваться ей будете только один раз. Действительно легче будет даже напечатать на пишущей машинке.

Получается, что одноразовые операции оптимизировать просто бессмысленно. Затраты в этом случае себя не окупают, поэтому не стоит тратить свои нервы на этот бессмысленный труд.

В самом начале этого раздела я раскритиковал вас как человека, который ленится хоть что-нибудь делать. Так вот, именно здесь вы можете проявить свою врожденную леность в полном объеме. В данном случае крутым считается не тот, кто целый день промучился и ничего не добился, а тот, кто выполнил свою работу быстрее и эффективнее. И эти две вещи путать нельзя.

Закон № 5

Нужно знать "внутренности" компьютера и принципы его работы. Чем лучше вы знаете, каким образом компьютер будет выполнять ваш код, тем лучше вы сможете его оптимизировать.

Этот закон относится только к программированию. Тут трудно привести полный набор готовых решений, но некоторые приемы я постараюсь описать.

- Старайтесь поменьше использовать вычисления с плавающей запятой. Любые операции с целыми числами выполняются в несколько раз быстрее.
- Операции умножения и, тем более, деления также выполняются достаточно долго. Если вам нужно умножить какое-то число на 3, то для процессора будет легче три раза сложить одно и то же число, чем выполнить умножение.

  А как же тогда экономить на делении? Вот тут нужно знать математику. У процессора есть такая операция, как сдвиг. Вы должны знать, что процессор думает в двоичной системе, и числа в компьютере хранятся именно в ней. Например, число 198 для процессора будет выглядеть как 11000110. Теперь посмотрим, как работают операции сдвига.

  Сдвиг вправо — если сдвинуть число 11000110 вправо на одну позицию, то последняя цифра исчезнет и останется только 01100011. Теперь введите это число в калькулятор и переведите его в десятичную систему. Ваш результат должен быть 99. Как видите — это ровно половина числа 198.

  Вывод: при сдвиге числа вправо на одну позицию вы делите его на 2.

  Сдвиг влево — возьмем то же самое число 11000110. Если сдвинуть его влево на одну позицию, то с правой стороны освободится место, которое заполняется нулем — 1 10001100. Теперь переведите это число в десятичную систему. Должно получиться 396. Что оно вам напоминает? Это 198, умноженное на 2.

  Вывод: при сдвиге числа влево вы умножаете его на 2.

Так что используйте сдвиги везде, где возможно, потому что эти операции работают в несколько раз быстрее умножения и деления (и даже сложения и вычитания).

- При создании функций (процедур) не обременяйте их большим количеством входных параметров. Перед каждым вызовом функции (процедуры) ее параметры прячутся в стек, а после выхода извлекаются оттуда. Чем больше параметров, тем значительнее расходы на общение со стеком.
- Тут же нужно сказать, что вы должны действовать аккуратно и с самими параметрами. Не вздумайте пересылать процедурам переменные, которые могут содержать данные большого объема в чистом виде. Лучше передать адрес ячейки памяти, где хранятся данные, а внутри процедуры работать с этим адресом. Вот представьте себе ситуацию, когда вам нужно передать текст размером в один том "Войны и мир"... Перед входом в процедуру программа попытается вогнать все это в стек. Если вы и не увидите его переполнения, то задержка точно будет значительная.
- В самых критических ситуациях (например, вывод на экран) можно воспользоваться языком ассемблера. Даже встроенный в C++ или Delphi ассемблер намного быстрее штатных функций языка. Ну а если скорость в каком-то месте уж слишком критична, то ассемблерный код можно вынести в отдельный модуль. Там его нужно откомпилировать с помощью компиляторов TASM или MASM и подключить к своей программе.

  Ассемблер — довольно быстрая и компактная вещь, но писать достаточно большой проект только на нем очень сложно. Поэтому я не советую им увлекаться, и использовать его только в самых критических по скорости местах.
- Не используйте длинных указателей. Что я под этим понимаю? Допустим, что у вас есть следующий код: `ClassName->SubClass->SubSubClass->Property`. Если в функции происходит несколько обращений к такому свойству, то это будет неэффективно. Каждый раз компилятор должен развертывать эту длинную цепочку для получения адреса переменной. Намного эффективнее будет завести локальную переменную в начале функции, присвоить ей значение и использовать локальную переменную, например:

```
void FuncName()
{
 int var = ClassName->SubClass->SubSubClass->Property;
 ...
 // Здесь уже используем var, что будет быстрее
 ...
}
```

Закон № 6

Для сложных расчетов можно подготовить таблицы с заранее определенными результатами и потом использовать эти таблицы в реальном режиме времени.

Когда появился первый Doom, игровой мир поразился качеству графики и скорости работы. Это действительно был шедевр программистской мысли, потому что компьютеры того времени не могли рассчитывать трехмерную графику в реальном времени. В те годы еще даже и не думали о 3D-ускорителях, и видеокарты занимались только отображением информации и не выполняли никаких дополнительных расчетов.

Как же тогда программистам игры Doom удалось создать трехмерный мир? Секрет прост. Игра не просчитывала сцены, все сложные математические расчеты были произведены заранее и занесены в отдельную базу (таблицу), которая запускалась при старте программы. Конечно же, занести все возможные результаты нереально, поэтому база хранила основные результаты. Когда нужно было получить значение, которого не было в заранее рассчитанной таблице, то бралось наиболее близкое число. Таким образом, Doom получил отличную производительность и достаточное качество 3D-картинки.

С выходом программы Quake игровой мир опять поразился качеству освещения и теней в сценах ее виртуального мира. Расчет света — очень сложная задача, не говоря уже о тенях. Как же тогда программисты игры смогли добиться такого качества сцен и в то же время высокой скорости работы игры? Ответ опять будет таким же — за счет таблиц с заранее рассчитанными значениями.

Через некоторое время игровая индустрия поразилась еще больше. Когда вышел Quake 3, в котором освещение рассчитывалось в реальном времени, то его мир казался искусственным, и даже Half-Life, который вышел позже и на движке старого Quake, выглядел намного естественнее и привычнее. Это получилось в результате того, что мощности компьютера не хватило для полных расчетов в реальном времени, а округления и погрешности пошли не на пользу игровому окружению.

И при всем при этом Quake всегда был легендарным и останется подлинным шедевром от настоящих хакеров.

Закон № 7

Лишних проверок не бывает.

Чаще всего оптимизация может привести к нестабильности исполняемого кода, потому что для увеличения производительности некоторые убирают ненужные, на первый взгляд, проверки. Запомните, что ненужных проверок

не бывает! Если вы думаете, что какая-то нестандартная ситуация может и не возникнуть, то она не возникнет только у вас. У пользователя, который будет эксплуатировать вашу программу, может возникнуть все что угодно. Он непременно нажмет на то, на что не нужно, или введет неправильные данные.

Обязательно делайте проверки всего того, что вводит пользователь. Делайте это сразу же, и не ждите, когда введенные данные понадобятся.

По возможности не выполняйте проверки в цикле, а выносите все за его пределы. Любые лишние операторы `if` внутри цикла очень сильно влияют на производительность.

Циклы — это слабое место любой программы, поэтому оптимизацию надо начинать именно с них. Внутри циклических операций не должно выполняться ничего лишнего, ведь это будет повторено много раз!

Закон № 8

Не переусердствуйте с оптимизацией. Слишком большие затраты на ускорение выполнения кода могут свести на нет приложенные усилия. Ставьте перед собой реальные цели и задачи.

Если у вас не получилось оптимизировать ваш код до необходимой степени, то нет смысла продолжать долгие попытки и мучения. Возможно, будет легче найти совершенно другой способ решения проблемы.

Не всегда получается добиться идеала, потому что оптимизация скорости и оптимизация качества — часто противоположные вещи. Лучше всего это заметно на примере программирования графики. Чтобы сцена (например, в компьютерных играх) выводилась на экран быстрее, можно сделать приближенные, но быстрые расчеты. Но тогда изображение получается невысокого качества. Поэтому очень часто приходится выбирать что-то одно.

В графических редакторах жертвуют скоростью, потому что тут не требуются расчеты в реальном времени. А вот в играх поступаться приходится качеством, иначе играть будет невозможно.

Закон № 9

Используйте ассемблер только там, где это действительно нужно.

Ассемблер позволяет использовать возможности компьютера максимально эффективно. Кода компилятор превращает код C/C++ в машинный код, то он волен решать, как будет выглядеть реализация в машинных инструкциях. Результат может быть очень быстрым и очень медленным, потому что язык программирования C абстрагирован достаточно хорошо, а C++ еще лучше. Если ничего не помогло, а производительность еще оставляет желать лучше-

го, то в этом и только в этом случае можно опуститься до ассемблерного кода и переписать медленные участки кода/функции на ассемблере. Но делать это нужно только тогда, когда другого выхода нет.

На мой взгляд, компилятор Visual C++ производства Microsoft очень хороший и прекрасно оптимизирует код. Не многие смогут написать код лучше и быстрее, поэтому лучше довериться опыту разработчиков C/C++.

Я до ассемблерных примеров спускаться не буду, ибо они выходят за пределы данной книги. Это достаточно сложная, но очень интересная тема. Если она вас интересует, то рекомендую обратиться к специализированной литературе по данной теме.

Можно использовать как встроенный в код ассемблер, так и внешний, и оба они могут дать хороший результат. Встроенный компилятор позволяет вам писать вставки на ассемблере непосредственно в коде на C++. Например, следующий код — простое присваивание с помощью ассемблера:

```
int number;

__asm mov number, 0;
```

В данном случае перед строкой, написанной на ассемблере, стоит `__asm`. Если нужно написать целый блок кода на этом языке, то его можно оформить следующим образом:

```
int number;

__asm
{
  mov bx, 0;
  mov number, bx;
}
```

Итог

Если вы прочитали этот раздел внимательно, то можете считать, что с азбукой оптимизации вы уже знакомы. Но это только основы, и тут есть куда развиваться. Я бы мог рассказать больше, но не вижу особого смысла, потому что оптимизация — это процесс творческий, и в каждой конкретной ситуации к нему можно подойти с разных сторон. И все же те законы, которые мы сегодня рассмотрели, действуют в 99,9% случаев.

Если вы хотите познать теорию оптимизации в программировании более глубоко, то вам нужно больше изучать принципы работы процессора и операционных систем. Главное, что законы вы уже знаете, а остальное придет со временем.

1.4. Безопасность кода

Безопасность чаще всего идет в разрез с оптимизацией, ибо выполнять код без каких-либо проверок корректности и защиты намного проще и быстрее. Зачем после каждой функции смотреть, корректно ли она отработала? Если файл открылся, то он есть и дальше уже можно с ним работать без каких-либо проверок, чтобы не тратить драгоценных тактов. Это серьезная ошибка, которая отрицательно сказывается на качестве создаваемого кода. Никогда не стоит доверять коду, который работает с внешними данными: файлами, сетью или вводимой пользователем информацией.

Во время написания кода нужно постоянно думать о безопасности. Если стоит выбор между производительностью и надежностью, то выбор должен однозначно падать на второе. Но при этом, нельзя защищать абсолютно все, и что нужно, и что не нужно (об этом мы поговорим в *разд. 1.4.2*). В большей степени это касается всех программ, которые работают с сетью, потому что в этом случае появляется опасность, что компьютер может быть взломан удаленно.

Пользовательские программы, которые работают локально, тоже должны быть надежны. Потребители не любят, когда софт падает и каждые пять минут сообщает об ошибке выполнения.

Почему эта тема так сильно касается нас? Мало того, что хакер должен писать не просто код, а качественный код, а для этого тот должен быть быстрым, надежным и безопасным. Да и, к тому же, мы будем работать все же с сетевыми приложениями, которые будут получать данные от удаленного компьютера, а что еще может быть опаснее?

Я ленивое создание, поэтому сам часто забываю проверять корректность данных, но постараюсь быть внимательным при написании примеров к книге. Но если где-то и что-то будет не так, то прошу простить старого и доверчивого программиста, который постоянно забывает о суровости нашей жизни, где очень много людей, готовых воспользоваться чужой ошибкой ради собственной наживы или популярности.

1.4.1. Планирование безопасности

О надежности и безопасности нужно думать постоянно и еще до того, как вы сели писать реальный код. Да, уже на этапе планирования необходимо определиться с тем, как программа будет гарантировать надежность.

Когда планируется создать программу, работающую с данными, передаваемыми по сети, вы заранее должны подумать о таких вещах:

- как сетевой трафик будет защищаться от изменения;
- насколько он важен, и нужно ли шифрование; если да, то какой алгоритм;

- как будет происходить идентификация и авторизация подключения;
- будет ли программа хранить пароли доступа; если да, то где и в каком виде.

Такие вещи не стоит оставлять на потом. Вы заранее должны знать, что может случиться с данными, кто является реальным врагом, и откуда этот враг может проникнуть в систему или помешать корректной работе программы.

Планируя программу и ее функции, всегда задавайте и отвечайте сами себе на вопросы: зачем, для чего, как эта функция отразится на безопасности, как сможет определенный метод защиты решить проблему безопасности. Для чего это делать? Вы должны четко понимать, для чего вы будете делать определенные шаги. Не нужно делать что-то только потому, что кто-то другой сделал так, ситуации бывают разные, поэтому в каждом отдельном случае нужно анализировать собственную ситуацию и принимать собственное решение.

Помимо этого необходимо определить все точки ввода информации пользователем или возможного поступления данных в программу. К таким точкам ввода я бы отнес:

- сетевые данные;
- файлы, особенно с текстом, который должен идти на выполнение (скрипты);
- данные, передаваемые пользователем через любые методы ввода информации (например, элементы управления).

На мой взгляд, самой жестокой средой является Интернет. Всемирная сеть является отражением нашей реальной жизни, потому что там есть как положительные герои, так и отрицательные. На мой взгляд, отрицательных намного больше, потому что многие считают, что их действия являются безнаказанными в сети. Хочется сказать, что это действительно так, но до поры до времени. Если хакер заинтересовал правоохранительные органы и не останавливается в своих действиях, то рано или поздно его все равно найдут.

На бога надейся, а сам не плошай, поэтому мы должны сами за себя стоять и защищать свою крепость. В данном случае нашей крепостью является программа, и мы будем говорить о ее защите.

Когда план готов, можно приступать к работе по реализации программы. Раньше я работал вообще без планов, но когда пару проектов провалилось в трубу только потому, что я неправильно поставил цели и неправильно определил решения, я стал больше уделять внимания планированию. Теперь первым делом я планирую проект, долго обдумываю его и только потом приступаю к реальному программированию конкретного решения.

Если проект большой, то каждые пару месяцев (можно и реже, это зависит от скорости разработки) лучше останавливаться и пересматривать/корректировать план. Все в нашей жизни меняется, и задание тоже может быть подвержено изменениям.

1.4.2. Уровень защиты

После того как вы определились, что будет защищаться, от кого и откуда появляются данные, необходимо определиться с уровнем защиты информации. Тут нужно ответить на вопрос, насколько критичной является потеря данных. Далеко не вся информация критична к потере. Как мы уже знаем, защита очень часто идет вразрез с производительностью, поэтому следует находить золотую середину. Это очень сложно, но от правильного решения будет зависеть соотношение производительности к безопасности и сам уровень безопасности.

Защита всей информации максимально возможным способом не эффективна: например, зачем абсолютно все данные шифровать сложнейшим алгоритмом? Шифрование — достаточно трудоемкий для процессора процесс. Необходимо определить реальные точки угрозы и необходимые, но достаточные методы защиты.

На мой взгляд, самые страшные ошибки — ошибки алгоритмов или логики, когда разработчик ошибочно выбрал метод защиты или неправильно реализовал алгоритм. Эти ошибки сложнее всего найти и чаще всего очень сложно исправить. Например, во время планирования сетевого протокола вы решили, что шифровать данные не имеет смысла, и посчитали, что пользователь будет использовать программу для передачи через Интернет не критичных к безопасности данных. Если пользователь захочет передать более критичные данные, то он может использовать VPN или другие методы навесной защиты.

Защита с помощью навесных методов хороша, но не всегда реализуема, поэтому у пользователя может возникнуть потребность в добавлении в программу возможности шифрования, хотя бы самого простого. Как это сделать? Это на словах все просто, но ведь на рынке уже существует множество копий вашей программы, которая не использует шифрования. Те пользователи, которые не обновятся до новой версии, не смогут работать с теми, у кого версия с шифрованием. Вы не можете заставить всех зайти на свой сайт и скачать обновление, поэтому придется реализовывать возможность работы в двух режимах, с шифрованием и без.

Я могу привести массу подобных примеров, в которых затраты на исправление просчетов планирования обходятся очень дорого и могут завести компанию в убытки.

1.4.3. Исправление ошибок

Я часто люблю говорить, что ошибки есть везде. Нет такой программы, которая была бы абсолютно безопасна и не содержала бы ошибок, если не считать самой гениальной разработки всех времен и народов — Hello World. Программист — человек, он склонен иногда совершать ошибки или заблуждаться. Я тоже человек и не могу знать все, и тоже совершаю ошибки. Но мы должны стремиться к тому, чтобы ошибок было как можно меньше и чтобы их проще было исправить.

Что я понимаю под простотой исправления? С одной стороны — простота, а значит и цена исправления зависит от качества кода и его оформления, если код написан ужасно и его тяжело читать, то искать и исправлять ошибку будет очень сложно. Существует множество методов оформления кода, чтобы он был читабельным. Кое-какие рекомендации вы сможете найти в дополнительной документации на компакт-диске к данной книге.

Советую также почитать книгу "Совершенный код" Макконнелла ("Питер", 2005). На мой взгляд, в ней есть недоработки и заблуждения, но это мое мнение, и я не имею права критиковать других, особенно публично, поэтому не буду говорить, что данное чтение обязательно.

Помимо качества кода на скорость и качество исправления ошибки влияет информирование пользователя и доставка исправлений потребителю вашей программы. Это также дорогое удовольствие для любой компании, поэтому чем проще и более отлаженным будет процесс загрузки обновлений, тем лучше. Обратите внимание, что за простоту обновлений борется не только Microsoft, но и Apple, Adobe, Sun Microsystem — все уважающие себя производители думают о пользователях.

Если даже вы не планируете создавать собственную компанию софтверного гиганта, заботиться о пользователе просто необходимо.

1.4.4. Шифрование

Этот раздел будет очень коротким, ибо смысл моего мнения в отношении шифрования прост: никогда не пытайтесь создавать собственный алгоритм шифрования. Лучшие умы мира работают над созданием стойких алгоритмов, которые не смогут взломать хакеры, а если смогут, то с сумасшедшими затратами. Поэтому забудьте идею собственного алгоритма. Лучше потратьте свои силы на создание более полезных вещей.

Второе: никогда не пытайтесь писать код реализации алгоритма. Допустим, что вы решили использовать существующий алгоритм и для этого собираетесь реализовать его в виде кода. Не нужно этого делать. Не менее умные программисты уже реализовали все необходимое в ОС и библиотеках сто-

ронних разработчиков. Я рекомендую довериться разработчику ОС. Вы не доверяете Microsoft? А зря, там работают очень умные люди, и даже если в реализации будет найдена ошибка, компания позаботится о том, чтобы исправить ее и доставить необходимые исправления конечному пользователю.

Не стоит думать, что вы сможете сделать код лучше, чем профессионалы, которые занимаются шифрованием и программированием алгоритмов уже долгое время. Вы можете поиграть с алгоритмами и попытаться что-то реализовать самостоятельно, но использовать этот код в боевых условиях я бы не стал.

Напоследок необходимо сказать о выборе алгоритма шифрования. Их достаточно много, и каждый обладает своими характеристиками скорости/стойкости/потребности в ресурсах. Я не могу выделить один, который подошел бы для всех случаев. Но я могу посоветовать обратить внимание на общепринятые стандарты RSA и DES.

Первая публикация алгоритм RSA была датирована 1977 г. Он основан на паре ключей, открытого и закрытого. Открытый ключ может публиковаться и быть доступным всем для шифрования данных и отправки этой информации владельцу. Только владелец, обладающий закрытым ключом, может расшифровать данные и увидеть текст, который был скрыт. Алгоритм RSA используется не только для шифрования данных, но и для создания цифровой подписи.

Алгоритм DES отличается в принципе, потому что он симметричен, т. е. используется только один ключ для шифрования и дешифрования. Он был разработан в недрах компании IBM и уже долгие годы является стандартом шифрования во многих странах, рождение этого алгоритма также датируется 1977 г.

При реализации защиты данных достаточно интересным вопросом является хранение ключа шифрования. Никогда, вообще никогда, ключ не должен быть прописан в коде программы или в жестко определенном месте. Какой смысл тратить усилия на шифрование, когда хакер может получить доступ к ключу? Пользователь должен иметь возможность самостоятельно выбирать положение ключа, и лучше будет, если ключ будет храниться на сменном носителе. Можно также использовать специализированное хранилище ОС.

1.4.5. Тестирование

На мой взгляд, тестирование должно происходить постоянно, и чем раньше оно начнется, тем лучше. Не все разделяют мою точку зрения, считая, что незачем тестировать программу, пока она не завершена и система безопасности не реализована полностью. Не могу согласиться с этим: готовая система

может просто скрыть дефекты, которые не будут видны до поры до времени. А в определенный момент или при изменениях в конфигурации уязвимость может выйти наружу.

Тестирование должно продолжаться и после выхода окончательного продукта на рынок. Будет лучше, если вы найдете уязвимость, а не взломщики. Изменить подпорченную репутацию — очень дорогое удовольствие. Будет лучше, если тестированием будут заниматься профессионалы. Большинство компаний раньше использовали метод массового тестирования, когда на рынок выбрасывался тестовый продукт, и его тестировали сами пользователи. Но результат не давал нужного качества, и финальные продукты, опробованные сотнями тысяч пользователей, продолжали содержать ошибки. Именно поэтому крупные игроки стали нанимать профессиональных людей, которые умеют находить ошибки, а это не такое уж распространенное умение.

1.4.6. Возможности системы

Максимально используйте возможности системы, особенно ее возможности безопасности. Это значит, что не стоит придумывать что-то новое, если это уже есть в системе. Я уже говорил в *разд. 1.4.4*, что не стоит писать собственную реализацию алгоритма шифрования, когда есть готовые и очень хорошие решения в ОС, которые можно использовать.

Помимо этого, я не стал бы разрабатывать собственную систему контроля доступа, когда есть ACL. Не стоит придумывать собственную систему аутентификации и авторизации, когда есть готовые решения.

Не стоит придумывать и собственную систему аудита, когда есть удобная система регистрации событий Windows, где можно сохранять информационные сообщения и сообщения об ошибках. Если вы решили создать собственную систему аудита, придется позаботиться о том, чтобы файлы журналов находились в безопасности и были защищены от атак.

1.4.7. Установка программы

Установка — это очень важный факт, которому нужно уделять достаточно серьезное внимание. Нельзя ставить программу, лишь бы она работала, или устанавливать абсолютно все в активном состоянии. Многие помнят, чем это заканчивалось. Компьютеры взламывали, причем пользователи даже не понимали, ведь ломали через сервисы, которые пользователям были не нужны.

Некоторые дистрибутивы Linux до сих пор сервисы сразу же устанавливают в автозапуск, что является недопустимым. Если пользователь выбрал установку, это не значит, что он тут же начнет использовать сервис. Возможно, он устанавливает его на будущее.

Программа должна запускаться только в минимальной конфигурации, чтобы выполнять базовые и самые необходимые функции. Все остальное должно быть отключено в установке по умолчанию, пока пользователь не выбрал иного. Только в этом случае пользователь осознает, что он делает, и отдает себе отчет в возможных проблемах. Это значит, что он будет следить за безопасностью включенной возможности.

Обратите внимание, что Windows (не только серверный, но и клиентский) уже устанавливается не в полной конфигурации, а только необходимой. Ничего лишнего. Например, если программа умеет работать с несколькими видами протоколов, то по умолчанию должен быть включен только один.

1.5. Распространенные уязвимости

Чтобы меньше совершать ошибок, нужно понимать, как хакеры могут использовать код для нарушения работы программы. Я выделил наиболее интересные и важные (на мой взгляд) проблемы безопасности, с которыми вы можете столкнуться.

1.5.1. Контроль данных

В *разд. 1.4.1* мы говорили о том, что нужно определить все точки входа в программу, где пользователь вводит данные или может их изменить. Программа, получив эти данные, перед использованием обязана проверить корректность данных. Вводя некорректную информацию, хакер может скомпрометировать систему или получить доступ к запрещенным данным.

Если вы работали с Web-приложениями, то должны были слышать о знаменитой атаке SQL Injection. Теоретически данная атака возможна и в клиентских приложениях, просто там чаще используют параметризированные запросы или хранимые процедуры, где инъекция проблематична.

В настоящее время Web-разработку больше ведут на более специализированных языках: PHP, ASP, Java и др. Я считаю, что программы нужно разрабатывать на том языке, который больше подходит для решения поставленной задачи. Для меня на C/C++ писать программы не очень удобно.

1.5.2. Переполнения

Для C++ наиболее распространенными, и в то же время наиболее опасными, являются ошибки переполнения, буфера стека и других не менее опасных мест. Дело в том, что в этом языке при выделении памяти нет никаких гарантий, что мы не выйдем за ее пределы.

Самое страшное, что все строки в C++ — это буферы памяти, которые также ничем не защищены. Работа со строками усложняется и тем, что существует множество различных кодировок, одних только Unicode есть только несколько вариантов.

Давайте посмотрим в теории, как происходит переполнение. На рис. 1.7 показан пример стека, состоящего из нескольких переменных. Последним в этом стеке находится адрес возврата из функции (не забываем, это просто теоретический пример псевдофункции). Программа при вызове функции помещает в стек все параметры, передаваемые в функцию, а также адрес возврата. Задача хакера поместить в буфер собственные данные и изменить адрес выполнения так, чтобы выполнить свой код.



Рис. 1.7. Стек до вмешательства хакера

Куда поместить код программы? У хакера не так уж и много вариантов, и самый простой — поместить его в стек. Проблема ОС в том, что код может выполняться не только из сегментах, но и из стека. Допустим, что `Переменная 1` является строковой и имеет размер в 10 Кбайт. Вполне реальная ситуация. Допустим, что значение строки вводит пользователь, передается по сети или читается из какого-то файла. И последнее допущение: функция копирования строки из источника в буфер `Переменная 1` не проверяет размер данных, а копирует все, что получает на вход. Все это вполне реальная задача, и подобные допущения не редкость, иначе мы не слышали бы о них так часто в бюллетенях безопасности.

Теперь у нас есть все необходимое. Мы должны в строке передать Shell-код или любой другой код, который мы хотим выполнить. Одно условие: этот код должен поместиться в отрезок памяти от начала `Переменная 1` до адреса возврата. Так как наша строка аж 10 Кбайт, то профессиональный хакер сможет

засунуть в этот участок слона, а если не сможет, то вызовет другую программу, которая сделает все, что необходимо злоумышленнику.

Теперь допустим, что от `Переменная 1` до адреса возврата около 20 Кбайт. Мы не будем знать точно, и это вполне реально. Мы знаем только приблизительный размер памяти. Как перезаписать адрес возврата так, чтобы он выполнил наш код? Все очень просто, мы должны сформировать следующий буфер:

```
Код хакера NOP NOP ... и так где-то двадцать килобайт ... адрес
```

После кода хакера идут нулевые операции NOP, которые процессор не выполняет, а просто переходит на следующую операцию. Там дальше стоит еще один NOP и т. д. Таким образом, мы стопроцентно перезапишем реальный адрес возврата NOP-кодом, а процессор, увидев это, будет скользить до конца буфера, где будет наш адрес возврата. Вот так вот, не зная точного положения адреса возврата, можно его перезаписать. Самое сложное тут — это узнать адрес Shell-кода. Но и это решаемая задача.

Код, который внедряется в буфер, будет выполняться от имени учетной записи, под которой выполняется сама программа (рис. 1.8).

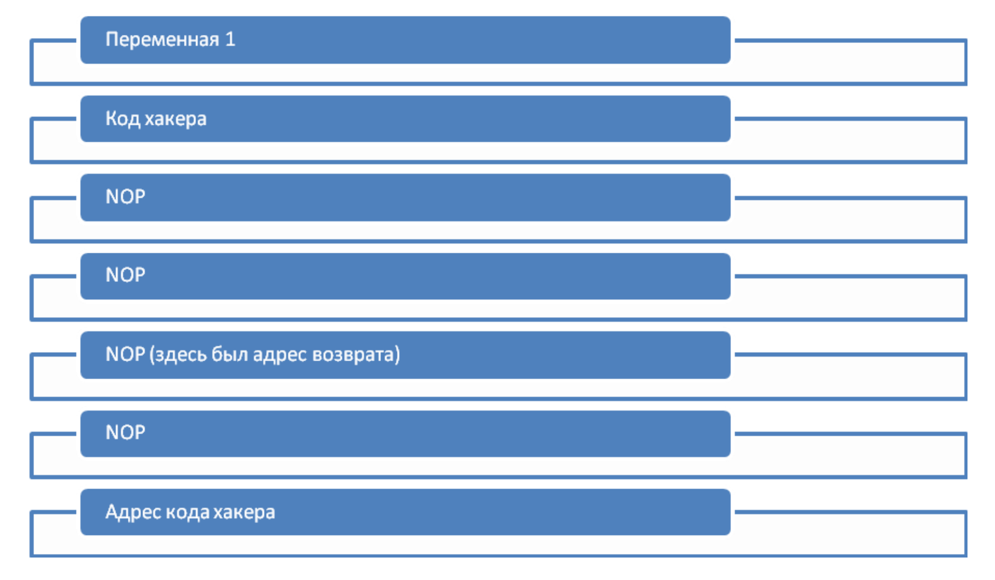


Рис. 1.8. Стек после вмешательства хакера

Реально опасный код может выглядеть следующим образом:

```
void Dangerous()
{
 char buf[1024];
```

```
 char input[2048];
 strcpy(buf, input)
}
```

В этом примере я копирую содержимое буфера `input` в локальный буфер `buf`. Проблема кроется в функции `strcpy`, которая просто не проверяет количество копируемых данных. Все что получено в переменной `input`, будет скопировано в `buf`, вне зависимости от размера приемника, и если приемник будет меньше, то его данные перепишутся и уничтожатся даже за пределами буфера.

Как решить проблему? Нужно проверить, достаточно ли в буфере приемника памяти, чтобы принять все входящие данные. Если нет, то можно скопировать только достаточный размер памяти, т. е. только 1024 байта, которые мы выделили для переменной `buf`, а можно сгенерировать сообщение об ошибке. Это уже зависит от конкретной ситуации.

Если вы используете Visual Studio, то он поможет вам создавать безопасные программы. Эта среда разработки по умолчанию будет отображать предупреждения, если вы решите использовать небезопасные функции. Например, на функцию `strcpy` компилятор выдаст предупреждение, показанное на рис. 1.9.

Error List

0 Errors | 1 Warning | 0 Messages

		Description	File	Line	Column	Project
⚠	1	warning C4996: 'strcpy': This function or variable may be unsafe. Consider using strcpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.	bufferoverflow.cpp	13		BufferOverflow

Рис. 1.9. Предупреждение об опасности функции

Это всего лишь предупреждение, а не ошибка, и исполняемый файл будет создан в любом случае. Просто его выполнение будет опасно. Вы можете отключить подобные сообщения, но я не рекомендую этого делать. Некоторые считают, что если самостоятельно сделать все необходимые проверки на размер буфера источника и приемника, то можно использовать и опасные функции. В принципе, это логично. Но неужели вам тяжело использовать безопасные функции? Это безопаснее! Если надеяться на свою внимательность, то иногда она может подвести, и последствия будут тяжелыми.

Сообщения о безопасности полезны и тем, что тут же предлагают вам возможный вариант решения проблемы. Например, сообщение на рис. 1.9 предлагает использовать функцию `strcpy_s` вместо `strcpy`. Я бы доверился Microsoft. Если у вас легальная копия, и вы получаете обновления, то в слу-

чае нахождения потенциальной угрозы в `strcpy_s` после обновления и перекомпиляции нам предложат новый, более безопасный вариант. Такие случаи уже были.

Безопасность не стоит на месте, и безопасная на сегодняшний день функция завтра может оказаться опасной. Да, такое бывает не очень часто, но все же. Если же вы будете использовать собственные варианты и надеяться на собственные силы, то никто не сможет вам помочь и подсказать. Только хакеры, когда найдут ошибку и воспользуются этим.

Именно из-за того, что безопасность не стоит на месте, я решил не описывать здесь все опасные функции, потому что тут вам поможет компилятор — узнать все опасное, а информация может устареть. Ограничившись основным, но самым необходимым, я предлагаю вам заглядывать на мои сайты **www.flenov.info**, **www.hackishcode.com**, а в ближайшее время, возможно, и **www.winvslin.com**, где я регулярно выкладываю что-то новое, свежее и интересное, в том числе и по безопасности. Замечу только, что опасными бывают и функции копирования текста, и форматирования (например, `printf`).

В самом конце я сообщу вам одну приятную новость: при компиляции в Visual Studio .NET с установленным флагом /GC переполнение буфера проблематично. Этот ключ предназначен именно для защиты и устанавливается по умолчанию для всех проектов, созданных в Visual Studio .NET 2003 и более поздних. Почему я сообщил это только сейчас? То, что уязвимость не удастся использовать, не значит, что программа стала распрекрасной. Ошибка как была, так и осталась, и хакер может навредить работе кода. Поэтому лучше все же использовать безопасные функции, которые не позволят хакеру выйти за пределы выделенного буфера.

Вместо ключа вы можете использовать оператор `#pragma`:

```
#pragma strict_gs_check(on)
```

Защита проста и в то же время очень эффективна. В стек перед адресом возврата добавляется случайное число. Перед выходом программа проверяет: если это число еще там и его никто не изменил, то выполнение можно продолжать, и стек не поврежден. Если же числа нет, то его могли уничтожить только переполнением буфера. Удаленный хакер не может догадаться, какое число могло быть записано в стек, поэтому любые попытки окажутся бесполезными, и максимум, чего он добьется — крушения программы, что тоже нехорошо.

Тут нужно еще заметить, что защита не будет добавляться, если:

- функция не содержит буферов, а значит, нечего защищать;
- функция имеет переменное число параметров;
- не включена оптимизация;

- функция начинается с ассемблерного кода в первой же строке;
- функция слишком мала (менее 4 байтов в размере).

Если эти условия не выполняются, то компилятор имеет право добавить защиту, которая представляет собой:

- код-пролог, который перед вызовом функции сохраняет в стеке случайное число безопасности;
- код-эпилог, который перед выходом из функции проверяет результат.

Если верить MSDN, то код-пролог выглядит следующим образом:

```
push        ebp                 ; Save ebp
mov         ebp, esp            ; Set stack frame pointer
sub         esp, localbytes     ; Allocate space for locals
push        <registers>         ; Save registers
```

Код-эпилог выглядит следующим образом:

```
pop         <registers>    ; Restore registers
mov         esp, ebp       ; Restore stack pointer
pop         ebp            ; Restore ebp
ret                        ; Return from function
```

Защита проста и в то же время не сильно уменьшает производительность системы. И все же, если какой-то код вызывается слишком часто, то данные также могут оказаться очень важными для программы. Если вы уверены, что ошибок нет, то можете явно указать компилятору, что не нужно добавлять защиты. Для этого к объявлению функции нужно добавить ключевое слово `naked`. Пример объявления функции без защиты:

```
__declspec( naked ) void func(параметры)
{
}
```

А может, вы захотите написать собственную защиту, и тут тоже поможет `naked`. Но я не вижу необходимости в этом, потому что разработчики Microsoft реализовали все очень неплохо.

В 64-битовых процессорах дело хакера осложнилось еще сильнее. ОС не помещает в стек адреса возврата из функций, а вместо этого хранит это значение в регистре процессора. Его просто так уже не изменишь, а выход за пределы буфера не даст такого эффекта. Но если хакер не сможет использовать уязвимость, это не значит, что он не сможет просто навредить программе.

А может быть, кто-то найдет метод переполнить буфер так, что изменит логику работы программы. Ведь когда-то никто не верил, что можно использо-

вать переполнение в куче. Это не стек, и там тоже нет адреса возврата, но факты взлома все же появились, и существует множество доказательств, что в куче тоже можно использовать уязвимость для выполнения произвольного кода.

Начиная с Windows Vista Beta 2, появилась еще одна защита, усложняющая переполнение буфера — Address Space Layout Randomization (ASLR случайное выравнивание адресного пространства). Сразу же скажу, что эта защита не может быть идеальным средством и не решит всех проблем сразу, это всего лишь один кирпичик в стене, защищающей вашу программу от хакера. Чем больше этих кирпичиков, тем надежнее система.

Что значит случайное выравнивание? Начнем с того, что до Windows Vista система всегда загружала программы с одним и тем же выравниванием не только на одном компьютере, но даже на разных. Если у двух компьютеров одинаковая версия Windows (в том числе — одной локализации, что является самым важным), то скорей всего после загрузки одной и той же программы адреса всех системных функций будут одинаковыми. Поэтому когда хакер внедряет свой код в буфер, написать Shell-код, который будет работать на разных компьютерах, не составляет особого труда.

Самое страшное, что и системные функции тоже загружали в одно и то же адресное пространство. Начиная с Vista Beta 2, адрес для загрузки выбирается случайным образом. Теперь Shell-коду будет труднее вызывать системные функции, потому что не так просто узнать их адреса.

В Интернете уже появились теоретические методы обхода данной защиты, но уж слишком теоретические они. На практике реализация достаточно проблематична. В двух словах, теория обхода заключается в том, что вариантов адресов для загрузки не так уж и много, всего 256, у вредоносного кода есть вероятность 1/256, что он сработает. Много это или мало, сказать трудно. Иногда даже самая маленькая вероятность может помочь хакеру, поэтому ASLR — это всего лишь кирпичик в защите, а не целая стена.

Для того чтобы в вашей программе адреса стали располагаться случайно, она должна быть собрана (link) с ключом /dynamicbase. В некоторых случаях эту функцию нужно будет, наоборот, отключать.

Защите программ в Microsoft уделяют в последнее время много внимания. Компания ищет не только свои собственные решения, но и смотрит на конкурентов. Не знаю, сама ли компания додумалась или взяла идею у BSD-систем, но в сборщике (linker) Visual Studio есть ключ /NXCOMPACT. Этот ключик устанавливается по умолчанию, если компонент требует подсистему 6.0 и выше (/SUBSYSTEM 6.0).

Суть золотого ключика в том, что нельзя выполнять изменяемые данные. Если область памяти помечена для выполнения и содержит код, то эти дан-

ные изменять нельзя. Если память помечена как область данных, то данные можно изменять, но они не могут быть выполнены. Даже если хакер поместит свой код в такую память, ее невозможно будет выполнить. А что, мне кажется это вполне логичным. Зачем выполнять код из памяти, которая должна хранить только данные? И как раньше до этого не догадались, ведь все необходимое есть уже давно на уровне железа.

Если у вас программа не самораспаковывающаяся (как UPX) и не самошифрующаяся, то этот ключ реально сможет поднять надежность программы и устойчивость к переполнению буфера.

Вот такой вот простой, но эффективный кирпичик в нашей стене безопасности может усложнить жизнь взломщика.

Очень интересный способ атаки был реализован в знаменитом черве CodeRed, который поражал IIS 4.0. Он использовал ошибку переполнения буфера, но при этом не пытался изменить адрес возврата из функции, а нарушал работу обработчика исключений стека, адрес которого также находится в стеке. При этом защита, которую предоставляет ключ /GS, оказывается бессильной.

Чтобы решить эту проблему, можно использовать ключ /SafeSEH. Данная защита не влияет на производительность и срабатывает только во время возникновения исключительной ситуации, поэтому грех не воспользоваться возможностями компилятора.

1.5.3. Ошибки логики

Это наверно самые страшные ошибки, потому что их очень сложно найти. На мой взгляд, они и самые опасные. Классическая проблема логики — выход за пределы массива, особенно в циклах. Посмотрите на следующий пример:

```
char buf[1024];
buf[sizeof(buf)] = '\0';
```

Все строки в C должны заканчиваться нулем. Здесь кто-то решил в качестве последнего символа массива установить нулевой символ, чтобы гарантировать, что конец буфера точно будет не за пределами выделенной области. А ведь действительно, есть функции, которые не гарантируют, что строка будет завершаться нулем. Тогда конец строки может оказаться где угодно в стеке или памяти. Это очень опасно, поэтому насильно устанавливать конец строки в последнем символе вполне логично.

Но проблема нашего кода в другом. Если вы уже заметили ошибку, то вы очень внимательны и опытны. Секрет ошибки в том, что все массивы нумеруются с нулевого индекса, а значит, последний символ определяется так:

```
buf[sizeof(buf)-1] = '\0';
```

Данная ошибка хоть и логическая, но ее можно отнести к переполнению буфера или можно назвать выходом за пределы буфера.

С функцией `sizeof` связана еще одна проблема безопасности — определение размера Unicode-строк. Хотите пример? И такой есть у меня:

```
_TCHAR buf[10];
printf("Buffer size: %d", sizeof(buf));
```

Угадайте, какой результат будет после выполнения программы? Мы завели буфер для хранения 10 символов, но функция `sizeof` возвращает размер буфера, а не количество символов. Если вам нужно определить количество символов, то следует разделить размер буфера, на размер одного элемента:

```
printf("Buffer size: %d", sizeof(buf) / sizeof(buf[0]));
```

Прежде чем вызывать функцию работы с Unicode-символами, точно определите, что она хочет, размер буфера для хранения строки или количество символов. Эта ошибка связана больше с переполнением буфером.

Ошибки логики бывают разные, и тут невозможно дать какие-то рекомендации. Например, логические операторы `if` (особенно вложенные) не обрабатывают всех возможных ходов. Вы можете подумать, что определенная ситуация никогда не наступит, и не напишете блок `else` для этой ситуации. Это только теоретически бывают случаи, которые никогда не наступают, поэтому лучше обрабатывать все возможные ветвления логики.

ГЛАВА 2



Простые шутки

Теперь можно приступать к написанию простых программ-приколов в Windows. Так как эта ОС самая распространенная, то и шутки в ней самые интересные. Думаю, что любой компьютерщик с удовольствием подкинет своему другу какую-нибудь веселую программку, которая введет жертву в легкий шок. В каждом из нас заложено еще при рождении стремление к превосходству. Все мы хотим быть лучшими, и программисты часто доказывают свое первенство с помощью написания чего-то уникального, интересного и вызывающего. Чаще всего в виде самовыражения выступают программы-шутки.

Хотя мои программы не будут вредоносными, но все же они должны быть кому-нибудь подброшены. Поэтому человека, которому должна быть подкинута программа, будем называть жертвой.

Большинство приколов этой главы основаны на простых функциях WinAPI. Хоть я и сказал, что начальные знания программирования желательны, но все же весь код я постараюсь расписывать очень подробно. Особый упор сделан на WinAPI-функции. Если некоторые возможности Visual C++ вы используете каждый день, то функции WinAPI можете применять достаточно редко (не считая основных), поэтому я даже не надеюсь, что вы знаете их все.

Я много раз встречал великолепных программистов, которые могли бы написать с закрытыми глазами любую программу для работы с базами данных, но при этом не могут программно переместить мышку. В этом нет ничего стыдного, просто их не интересовала эта тема. Стоит только задаться вопросом, и я уверен, что они нашли бы решение.

По сравнению с первым изданием я постарался сжать информацию, чтобы программистам с опытом не так скучно было читать эту главу. Здесь все же будут описываться достаточно простые вещи. Но если вы начинающий программист, учиться на таких примерах будет очень интересно, и я уверен, что

вы полюбите программирование, потому что это весьма даже захватывающее занятие.

2.1. Летающий *Пуск*

Вспоминаю, как я первый раз увидел Windows 95. Мне так понравилась кнопка **Пуск**, что я полюбил ее до глубины **Выключить компьютер**. Вскоре в нашем институте обновили парк машин, и на них тоже поставили Windows 95. Мне так захотелось подшутить над своими однокурсниками, что я решил написать программку, которая подбрасывала бы кнопку **Пуск**. Сказано — сделано, написал (тогда я любил Delphi и использовал его) и запустил на всех машинах. С каждым взлетом кнопки **Пуск** ламеры испуганно взлетали вместе с ней. А через некоторое время я увидел и в Интернете подобный прикол.

Сейчас я повторю свой старый подвиг и покажу вам, как самому написать такую программу с использованием Visual C++. Так что усаживайтесь поудобнее, наша кнопка **Пуск** взлетает на высоту 100 пикселов!

В этом примере мы пойдем на небольшую хитрость и будем подбрасывать не саму кнопку **Пуск**, а окно, в котором будет нарисовано изображение кнопки. Да, это обман, но иногда можно пойти на это, особенно если результат будет стоящий. Чуть позже мы рассмотрим пример реального доступа к системной кнопке, а пока ограничимся таким трюком, потому что это даже интересней.

Но прежде чем начать, нужно подготовить картинку с изображением кнопки **Пуск**. Для этого вы можете нарисовать ее своими руками в любом графическом редакторе. Ну а если вы IBM-совместимый человек, то можете нажать клавишу <PrintScrn> (или <PrintScreen>), чтобы запомнить изображение экрана в буфере обмена, а потом выполнить вставку содержимого буфера в любом графическом редакторе. Далее простыми манипуляциями вырезать изображение кнопки и сохранить его в отдельном файле.

У меня получилась картинка размером 50×20, и вы найдете ее на компакт-диске в каталоге Demo/Chapter2/Start Button/Start.bmp. Можете воспользоваться этим файлом.

Создайте новый проект в Visual C++ типа Win32 Project с именем Start Button и добавьте в него нашу картинку. Для этого откройте ресурсы, дважды щелкнув по файлу Start Button.rc. Перед вами откроется окно с деревом ресурсов (рис. 2.1).

Щелкните в этом окне правой кнопкой мыши и в появившемся выпадающем меню выберите пункт **Add resource**. Вы должны увидеть окно добавления ресурсов с возможностью выбора типа создаваемого ресурса (рис. 2.2). В этом окне выберите пункт **Bitmap** и нажмите кнопку **New**.

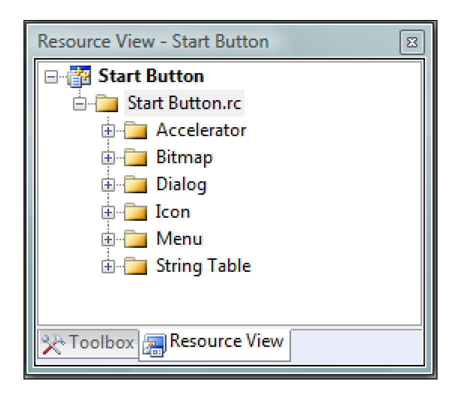


Рис. 2.1. Окно просмотра ресурсов

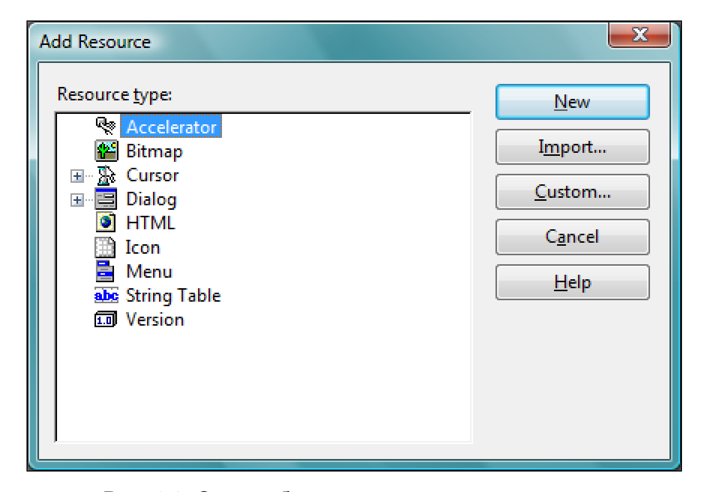


Рис. 2.2. Окно выбора типа создаваемого ресурса

В этом разделе будет создан новый ресурс для хранения изображения. Под окном просмотра ресурсов вы можете видеть окно свойств изображения (рис. 2.3). Здесь нужно первым делом изменить свойство **Colors** (Количество цветов), установив значение **256 Color** или **True Color**. Ширину и высоту (свойства **Width** и **Height**) нужно указать в соответствии с вашей картинкой.

Откройте изображение кнопки **Пуск** в любом графическом редакторе (например, Paint) и скопируйте его в буфер обмена (чаще всего для этого нужно выделить изображение и выбрать меню **Edit/Copy**). Вернитесь в редактор Visual C++ и выполните команду **Edit/Paste**. Вы должны увидеть нечто похожее на рис. 2.4.

Теперь переходим непосредственно к программированию. На этом примере мы рассмотрим некоторые приемы, которые будем использовать в дальнейшем достаточно часто.

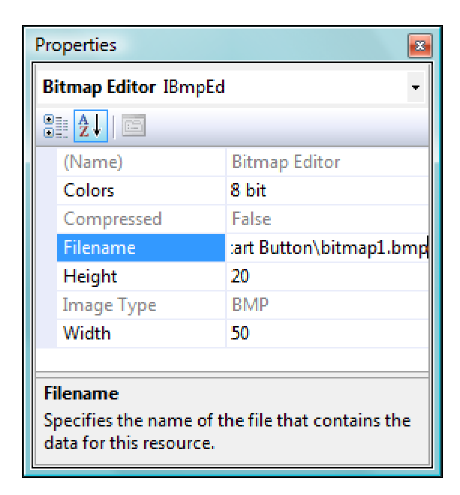


Рис. 2.3. Окно свойств изображения

Рис. 2.4. Изображение кнопки **Пуск** в редакторе ресурсов

Откройте файл Start Button.cpp. Для этого найдите его в окне **Solution Explorer** в разделе **Source Files** и дважды щелкните на строке с именем. В самом начале файла найдите раздел глобальных переменных, который начинается с комментария `Global Variables:`. После этого комментария добавим две переменные:

```
// Global Variables:
HWND hWnd;
HBITMAP startBitmap;
```

Первая переменная — `hWnd` — имеет тип `HWND`, который используется для хранения указателей на окна. В ней мы и сохраним указатель на созданное в примере окно, чтобы в любой момент можно было получить к нему доступ. Вторая переменная — `startBitmap` — имеет тип `HBITMAP`, который используется для хранения картинок, и мы поместим сюда наше изображение кнопки **Пуск**.

Теперь переходим в функцию `_tWinMain`. В ней, после загрузки из ресурсов текста окна и имени класса окна, добавим следующую строчку кода:

```
startBitmap = (HBITMAP)::LoadImage(hInstance,
   MAKEINTRESOURCE(IDB_BITMAP1), IMAGE_BITMAP,
   0, 0, LR_DEFAULTCOLOR);
```

Здесь мы переменной `startBitmap` присваиваем загруженную из ресурсов картинку. Для этого вызывается функция `LoadImage`, которой (в скобках) нужно передать следующие параметры:

- экземпляр приложения — переменная `hInstance`, которую мы получили в качестве первого параметра нашей функции `_tWinMain`, именно она содержит необходимое значение экземпляра;
- имя ресурса — наша картинка сохранена под именем `IDB_BITMAP1`;
- тип изображения — в нашем случае растровая картинка — `IMAGE_BITMAP`;
- размеры (следующие два параметра) — мы указали значение 0, чтобы использовать текущие размеры картинки;
- флаги — здесь указано `LR_DEFAULTCOLOR`, что означает использование цветов по умолчанию.

Больше в этой функции мы пока ничего изменять не будем. Чуть позже мы еще сюда вернемся и добавим пару строк, а сейчас переходим в функцию `InitInstance`. Перед выходом из функции (до строки с оператором `return`) нужно написать код из листинга 2.1.

Листинг 2.1. Обновленная функция `InitInstance`

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    ...
    ...
```

```
    // Следующие строки добавлены нами
    int Style;
    Style = GetWindowLong(hWnd, GWL_STYLE);
    Style=Style || WS_CAPTION;
    Style=Style || WS_SYSMENU;
    SetWindowLong(hWnd, GWL_STYLE, Style);

    return TRUE;
}
```

Код, добавленный нами, начинается с объявления переменной `Style`, которая будет иметь тип `int` (целое число). В следующей строке этой переменной присваивается результат выполнения функции `GetWindowLong`. Она возвращает настройки окна и в скобках нужно передать два значения:

- окно, параметры которого необходимо узнать, — мы указываем только что созданное нами окно;
- тип параметров — нас будет интересовать стиль окна, поэтому указана константа `GWL_STYLE`.

Зачем нам нужен стиль? Просто окно по умолчанию имеет заголовок, кнопки максимизации и минимизации, а нам все это не нужно. Для этого из полученного стиля в следующих двух строках удаляется заголовок окна и системное меню, которое содержит кнопки.

Теперь выполняем функцию `SetWindowLong`, которая записывает значения обратно в настройки окна. Если сейчас запустить программу, то вы увидите только клиентскую часть — серый квадрат без заголовка, кнопок и обрамления.

Переходим в функцию `WndProc`, где у нас обрабатываются все события. Нас будет интересовать рисование, поэтому добавим следующий обработчик события `WM_PAINT`, а код этого обработчика можно увидеть в листинге 2.2.

Листинг 2.2. Функция `WndProc` с обработчиком для рисования

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        int wmId, wmEvent;
        PAINTSTRUCT ps;
        HDC hdc;
        HDC hdcBits;

        switch (message)
        {
```

```
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                // TODO: Add any drawing code here...
                Rectangle(hdc, 1,1,10,10);
                        hdcBits=::CreateCompatibleDC(hdc);
                        SelectObject(hdcBits,startBitmap);
                        BitBlt(hdc, 0, 0, 50, 20, hdcBits, 0, 0, SRCCOPY);
                        DeleteDC(hdcBits);
                EndPaint(hWnd, &ps);
                break;
        ...
        // Продолжение кода функции, где я опустил
        // сгенерированные средой обработчики
        ...
        }
        return 0;
}
```

Чтобы начать рисование, надо знать, где мы будем это делать. У каждого окна есть контекст, в котором можно рисовать средствами Windows. Чтобы получить его для текущего окна, нужно вызвать функцию `BeginPaint`. Эта функция как раз и вернет нам указатель на контекст окна, указанного в качестве первого параметра.

Но чтобы отобразить изображение нашей кнопки **Пуск**, надо еще подготовить картинку. В WinAPI нет готовой функции для рисования растрового изображения, но есть возможность выбрать изображение в контекст и возможность копирования между контекстами. Для этого сначала надо создать контекст рисования, совместимый с тем, что использует окно, чтобы можно было без проблем производить копирование. Воспользуемся функцией `CreateCompatibleDC`, которой нужно передать контекст окна, а она нам вернет новый контекст, совместимый с указанным.

Следующим шагом мы должны выбрать в новый контекст нашу картинку. Для этого можно вызвать функцию `SelectObject`, у которой два параметра:

- контекст, в который нужно выбрать объект, — указываем созданный нами контекст, на основе оконного;
- объект, который надо выбрать, — указываем картинку.

Вот теперь можно производить копирование с помощью функции `BitBlt`. У этой функции нужно указать следующие параметры:

- контекст рисования, в который надо копировать (приемник), — указываем контекст окна;

- следующие четыре параметра являются левой верхней координатой, шириной и высотой прямоугольника, в который надо скопировать изображение (целые числа). В данном случае левая и верхняя позиции будут равны нулю, чтобы картинка располагалась в левом верхнем углу окна. Ширина и высота равны размеру картинки (50×20);
- источник копирования — указываем контекст `hdcBits`, в котором находится наша картинка;
- следующие два параметра задают левую верхнюю координату прямоугольника в контексте-источнике (именно от этой точки будет взято изображение для копирования) — указываем нули, т. к. нас интересует вся кнопка, начиная с левого верхнего угла;
- последний параметр указывает на тип копирования — используем флаг `SRC_COPY`, т. к. будем создавать копию источника в приемнике.

После рисования нам уже не нужен контекст, который мы создали для картинки, и хорошим тоном было бы удалить его. Для этого вызываем функцию `DeleteDC` и в качестве параметра указываем наш контекст рисования.

Завершаем рисование вызовом метода `EndPaint`. Таким образом, мы ставим точку в начатом функцией `BeginPaint` рисовании.

Теперь в нашем окне в левом верхнем углу будет рисоваться изображение кнопки **Пуск**. Остается сделать самую малость — уменьшить размер окна до размеров изображения, чтобы пользователь видел только картинку, и заставить окно двигаться. Для этого мы должны написать функцию `DrawStartButton` (листинг 2.3), желательно, до функции `_tWinMain`.

Листинг 2.3. Функция, заставляющая окно двигаться

```
void DrawStartButton()
{
 int i;
 int toppos=GetSystemMetrics(SM_CYSCREEN)-23;

 // Устанавливаем верхнюю позицию окна в левый нижний угол экрана
 SetWindowPos(hWnd, HWND_TOPMOST, 4, toppos, 50, 20, SWP_SHOWWINDOW);
 UpdateWindow(hWnd);

 // Сейчас будем поднимать кнопку
 // От 1 до 50 выполнять действия для изменения положения окна
 for (i=0; i<50; i++)
 {
   toppos=toppos-4;
```

```
    SetWindowPos(hWnd, HWND_TOPMOST, 4, toppos, 50, 20, SWP_SHOWWINDOW);
    Sleep(h,15);// Задержка в 5 миллисекунд
  }

  // Опускаем кнопку вниз
  for (i=50; i>0; i--)
  {
    toppos=toppos+4;
    SetWindowPos(hWnd, HWND_TOPMOST, 4, toppos, 50, 20, SWP_SHOWWINDOW);
    Sleep(h,15);// Задержка в 5 мс
  }
}
```

Чтобы правильно расположить окно с нашей кнопкой на экране компьютера, мы должны знать его разрешение. Для этого выполняется следующая строка кода:

```
int toppos = GetSystemMetrics(SM_CYSCREEN) — 23;
```

Здесь вызывается функция `GetSystemMetrics`, которая возвращает значение определенного системного параметра. В скобках указывается параметр, который нас интересует (в данном случае `SM_CYSCREEN`, высота экрана). Из результата вычитаем число 23 (высота картинки плюс еще 3 пиксела) и сохраняем результат в переменной `toppos`.

Таким образом, мы вычислили верхнюю позицию окна с изображением кнопки и можем его туда переместить. Еще необходимо, чтобы наше окно всегда было поверх остальных. Обе эти операции можно сделать, вызвав только одну функцию `SetWindowPos`. У нее 7 параметров:

- окно, которое надо переместить, — указываем наше окно;
- место размещения (после какого окна нужно расположить указанное) — устанавливаем флаг `HWND_TOPMOST` (поверх всех);
- следующие четыре параметра определяют прямоугольник, в котором должно располагаться окно. Левую позицию задаем равной 4. Верхнюю — равной переменной `toppos`. Ширина и высота окна должны определяться размерами картинки. Возможно, что после запуска программы вам придется подкорректировать левую верхнюю позицию в зависимости от подготовленной вами картинки;
- последний параметр задает режим отображения окна — устанавливаем флаг `SWP_SHOWWINDOW` (просто отобразить).

После этого прорисовываем окно в новой позиции с помощью вызова функции `UpdateWindow(hWnd)`. В скобках указано окно, которое надо отобразить.

Теперь наше окно расположено в нужном месте, и можно приступить к его анимации (движению по экрану). Для этого запускаем цикл от 0 до 50, внутри которого выполняются следующие действия:

```
for (i=0; i<50; i++)
{
       toppos=toppos-4;
       SetWindowPos(hWnd, HWND_TOPMOST, 4, toppos, 50, 20,
          SWP_SHOWWINDOW);
}
```

Сначала уменьшается значение переменной `toppos` на четыре пиксела. Таким образом, окно будет ползти вверх по экрану. Потом перемещаем это окно в новую позицию. Чтобы движение кнопки не было слишком быстрым, после перемещения окна делаем задержку в 15 мс.

Итак, наш цикл двигает кнопку вверх по экрану. После этого мы должны вернуть кнопку на место, для чего запускается еще один цикл, в котором тем же способом кнопка движется в обратном направлении.

Теперь у нас все готово, и мы должны вернуться в функцию `_tWinMain` и написать там вызов функции `DrawStartButton`. Я рекомендую сделать этот вызов перед циклом обработки сообщений и внутри него:

```
DrawStartButton();

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
  DrawStartButton();
  if (!TranslateAccelerator(msg.hwnd, hAccelTable,&msg))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

Теперь при старте программы наша лжекнопка будет взлетать и возвращаться на место поверх настоящей кнопки. Если вы попытаетесь навести на настоящую кнопку мышь, то она пройдет поверх окна лжекнопки, и наша программа получит сообщение от мыши, а значит, выполнится цикл обработки событий, в котором функция `DrawStartButton` снова подбросит нашу кнопку на некоторую высоту.

Результат работы программы можно увидеть на рис. 2.5. Ничего разрушительного в этом нет, но вашим друзьям понравится.

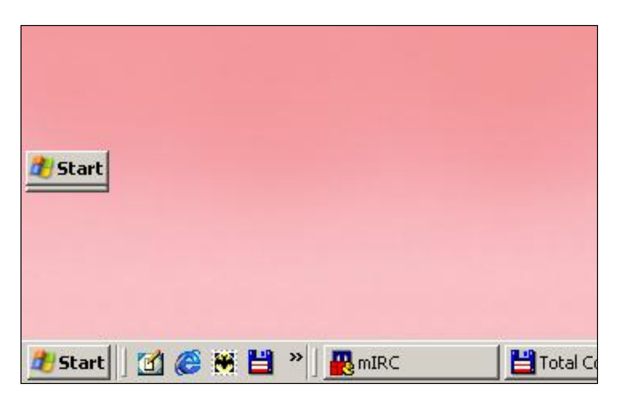


Рис 2.5. Пример работы программы



Исходный код и запускаемый файл этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter2\Start Button. Чтобы запустить программу, выберите меню **Debug/Start**.

Очень много шуток могут основываться на эффекте "подставки", когда мы рисуем или двигаем не сам объект, а совершенно другое окно с изображением нужного объекта. Таким образом, можно даже просто вывести на экран диалоговое окно с сообщением "Форматировать диск C:", и пользователь может не на шутку испугаться. Ну а если вывести десять сообщений с надписью "Virus Alert!!!", то даже профессионал может поверить.

Вспоминаются времена 90-х годов, когда даже вирусы были веселыми. Все их действия заключались в том, что они выводили веселые сообщения, играли музыку через PC Speaker или выводили на экран какую-то ASCII-графику. При этом самое страшное, что они делали — копировались без спроса между компьютерами. Конечно же, даже эти вирусы нельзя считать хорошими, но они, по крайней мере, были с изюминкой. Нынешние вирусы не несут в себе вообще ничего пристойного и интересного.

2.2. Начните работу с кнопки *Пуск*

Если вы сами устанавливали Windows версии 9*x*, то после первого запуска, наверное, видели сообщение ОС типа "Начните работу с этой кнопки" и стрелку, указывающую на кнопку **Пуск**. Я достаточно долго работал администратором сети, и мне наскучило отвечать пользователям на вопрос: "А где у меня программа XXX?". После очередного вопроса я написал программу, которая постоянно открывает меню, появляющееся по нажатию кнопки **Пуск**. Сейчас нам предстоит написать подобный пример.

Создайте новое приложение Win32 Project. Я назвал новый проект CrazyStart, но вы можете назвать и по-другому. В данном примере имя проекта не будет использоваться, и путаницы в понимании не будет.

Откройте файл с кодом вашего проекта, он должен иметь имя вашего проекта и расширение cpp (у меня это CrazyStart.cpp). Найдите функцию `_tWinMain` и доведите ее до вида, как в листинге 2.4. По комментариям, которые указаны в листинге, вы легко можете определить, что нужно добавить. Добавление нужно начать после комментария "Main message loop".

Листинг 2.4. Функция `_tWinMain`

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
   ...
   ...

   // Main message loop:
   // Необходимо добавить в свой код следующие три строки:
   HWND  hTaskBar, hButton;

   hTaskBar= FindWindow("Shell_TrayWnd",NULL);
   hButton= GetWindow(hTaskBar, GW_CHILD);

   while (GetMessage(&msg, NULL, 0, 0))
   {
      if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
      {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
      // Нажать кнопку "Пуск"
      // Необходимо добавить в свой код следующие две строки:
      SendMessage(hButton, WM_LBUTTONDOWN, 0, 0);
      Sleep(1000);
   }
   return (int) msg.wParam;
}
```

Сначала мы объявляем две переменные: `hTaskBar` и `hButton` типа `HWND`. Это уже знакомый нам тип, который используется для ссылки на окна. Потом мы

выполняем функцию FindWindow, которая ищет окно по заданным двум параметрам:

- имя класса окна — это имя используется при регистрации окна в системе;
- имя заголовка окна — текст, который указан в заголовке.

Кнопка **Пуск** расположена на Панели задач, которая является окном, и именно его мы хотим найти. Класс этого окна — Shell_TrayWnd, что мы и указываем в первом параметре. Заголовка нет, поэтому и имени окна не будет, так что второй параметр пустой и равен NULL.

На Панели задач есть только одна кнопка — **Пуск**, поэтому мы можем получить на нее ссылку с помощью вызова функции GetWindow. Эта функция имеет два параметра:

- указатель на окно;
- "родственные связи" искомого окна и указанного. У нас кнопка находится на окне, поэтому окно является для нее родителем, а сама кнопка — подчиненным, и мы должны указать флаг GW_CHILD.

Таким образом, мы получим указатель на кнопку и сохраним его в переменной hButton. В цикле обработчика сообщений мы посылаем кнопке **Пуск** сообщение с помощью функции SendMessage со следующими параметрами:

- окно, сообщение которому мы хотим послать, — указатель на кнопку **Пуск**;
- сообщение, которое надо послать, — отсылаем WM_LBUTTONDOWN, что равносильно нажатию левой кнопки мыши. Именно такое событие получает окно, когда нажимается кнопка.

Кнопка **Пуск**, получив наше сообщение, будет "думать", что по ней щелкнули левой кнопкой мыши, и отобразит меню.

После этого вызывается функция Sleep, которая делает задержку в заданное количество миллисекунд. У нас указано 1000, что равносильно одной секунде. Я люблю использовать для задержек функцию WaitForSingleObject. Чтобы сделать задержку этой функцией, можно использовать следующий код:

```
HANDLE h = CreateEvent(0, true, false, "et");
WaitForSingleObject(h, 15); // Задержка в 15 мс
CloseHandle(h);
```

Кода тут немного больше, но ничего сложного. В первой строке создается событие с помощью функции CreateEvent. Во второй строке вызывается функция ожидания событий. Так как его никто и никогда не установит, кроме нас, то функция будет ожидать максимум времени, а максимум — это второй параметр, т. е. 15 мс. В третьей строке мы закрываем описатель события.



Исходный код и запускаемый файл этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter2\CrazyStart.

2.3. Светомузыка над кнопкой *Пуск*

Над кнопкой **Пуск** можно издеваться достаточно долго. Еще одна шутка, которую можно сделать с этой кнопкой, — спрятать ее.

Для следующей задачи вы можете создать новое приложение или воспользоваться кодом из предыдущего примера, немного подкорректировав функцию `_tWinMain` (листинг 2.5).

Листинг 2.5. Обновленная функция `_tWinMain`

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
    ...

    // Main message loop:
    HWND  hTaskBar, hButton;

    hTaskBar= FindWindow("Shell_TrayWnd",NULL);
    hButton= FindWindowEx(hTaskBar, 0,"Button", NULL);

    while (GetMessage(&msg, NULL, 0, 0))
    {
       if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
       {
           TranslateMessage(&msg);
           DispatchMessage(&msg);
       }
       // Спрятать кнопку "Пуск"
       ShowWindow(hButton, SW_HIDE);
       // Насладимся зрелищем 2 секунды
       Sleep(50);
       // Показать кнопку "Пуск"
       ShowWindow(hButton, SW_SHOW);
       Sleep(50);
    }
```

```
    return (int) msg.wParam;
}
```

В этом примере мы точно так же ищем окно Панели задач и кнопку **Пуск** на ней. Отличие от предыдущего примера скрыто внутри обработчика событий. Здесь мы используем функцию `ShowWindow`. В *главе 1* мы уже рассматривали эту функцию и знаем, что она предназначена для отображения окна. Но она может быть использована и для того, чтобы максимизировать, минимизировать или спрятать окно.

Кнопки в Windows — это те же окна, поэтому мы можем использовать эту функцию для нашей кнопки **Пуск**. Функция `ShowWindow` вызывается два раза, и оба раза первый параметр передается в виде указателя на найденную кнопку. В качестве второго параметра первый раз передаем флаг `SW_HIDE`, который заставляет кнопку спрятаться, а во второй раз — `SW_SHOW`, чтобы отобразить кнопку. Между вызовами функции `ShowWindow` стоит функция `sleep`, которая выполняет задержку для того, чтобы пользователь успел увидеть панель с кнопкой и без нее.

Запустите программу, и она будет в бесконечном цикле прятать и отображать кнопку **Пуск**. Теперь вы можете без проблем написать код, который просто прячет главную кнопку Windows, и пользователь больше не сможет на нее нажать.

Еще одно отличие этого примера, здесь кнопка на Панели задач ищется иначе. Если раньше мы использовали `GetWindow`, то в этом примере применяется функция `FindWindowEx`. Она схожа с `FindWindow`, но позволяет производить более точный поиск не только главных окон, но и дочерних, принадлежащих другим окнам, потому что содержит следующие параметры:

- окно, на котором нужно искать элемент управления, — благодаря этому параметру мы можем искать кнопку внутри окна;
- элемент управления на этом окне, с которого нужно начинать поиск, — если здесь указать 0, то поиск будет начинаться с самого первого элемента управления;
- класс элемента управления — в нашем случае это кнопка, значит, нужно указать `Button`;
- имя — если указать нуль (`NULL`), то будет происходить поиск всех элементов подобного класса.



Исходный код и запускаемый файл этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter2\StartMusic.

Немного изменив код, можно сделать светомузыку из всей Панели задач. Необходимый код вы можете увидеть в листинге 2.6. Правки минимальны, просто вместо кнопки в функцию `ShowWindow` отправляется указатель на Панель задач.

Листинг 2.6. Светомузыка для Панели задач

```
int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
      LPTSTR lpCmdLine,  int nCmdShow)
{
    ...
    ...

    HWND  hTaskBar;

    hTaskBar= FindWindow("Shell_TrayWnd",NULL);

       // Main message loop:
       while (GetMessage(&msg, NULL, 0, 0))
       {
              if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
              {
                     TranslateMessage(&msg);
                     DispatchMessage(&msg);
              }
          // Спрятать задачи
          ShowWindow(hTaskBar, SW_HIDE);
          // Насладимся зрелищем 2 с
          Sleep(100);
          // Показать задачи
          ShowWindow(hTaskBar, SW_SHOW);
          Sleep(100);
       }

       return (int) msg.wParam;
}
```

Примечание

Исходный код и запускаемый файл этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter2\Tasks.

2.4. Продолжаем шутить над Панелью задач

Как мы уже знаем, Панель задач — это такое же окно, и над ним можно жестоко издеваться всеми доступными функциями для работы с окном. Когда мы писали первый пример, который подбрасывал лжекнопку, то мы поднимали свое окно с изображением кнопки **Пуск**. Кто нам теперь мешает модифицировать этот пример и подбросить реальную кнопку **Пуск**, когда мы уже знаем, как получить к ней доступ.

Но не все так просто, и сейчас мы рассмотрим пример, в котором увидим несколько интересных приемов, позволяющих шутить над реальной кнопкой **Пуск**.

Создайте новый проект StartEnable и добавьте в раздел глобальных переменных следующие строки:

```
HWND hWnd;
HWND  hTaskBar, hButton;
HMENU MainMenu;
```

В этом месте мы объявляем три переменные, которые будут ссылаться на окна:

- `hWnd` — будет хранить указатель на наше окно, чтобы мы могли его использовать в любой точке кода;
- `hTaskBar` и `hButton` — будут хранить указатели на Панель задач и кнопку **Пуск**;
- `MainMenu` — сюда мы загрузим меню нашей программы, чтобы в дальнейшем использовать его.

Теперь переходим в функцию `_tWinMain` и в ней добавляем код из листинга 2.7 до цикла обработки событий.

Листинг 2.7. Код, который нужно добавить в функцию `_tWinMain`

```
hTaskBar = FindWindow("Shell_TrayWnd",NULL);
hButton = GetWindow(hTaskBar, GW_CHILD);
MainMenu = LoadMenu(hInstance, (LPCTSTR)IDC_STARTENABLE);

SetParent(hButton, 0);

int i;
HANDLE h;
int toppos=GetSystemMetrics(SM_CYSCREEN)-23;
```

```
// Установить верхнюю позицию окна в левый нижний угол экрана
SetWindowPos(hButton, HWND_TOPMOST, 4, toppos, 50, 20,
  SWP_SHOWWINDOW);
UpdateWindow(hButton);
// Создать пустой указатель h, который будет использоваться для задержки
h=CreateEvent(0, true, false, "et");

// Сейчас будем поднимать кнопку
// Цикл по изменению позиции кнопки
for (i=0; i<50; i++)
{
  toppos=toppos-4;
  SetWindowPos(hButton, HWND_TOPMOST, 4, toppos, 50, 20,
           SWP_SHOWWINDOW);
  WaitForSingleObject(h,15);  // Задержка в 5 мс
}
for (i=50; i>0; i--)
{
   toppos=toppos+4;
   SetWindowPos(hButton, HWND_TOPMOST, 4, toppos, 50, 20,
           SWP_SHOWWINDOW);
   WaitForSingleObject(h,15);// Задержка в 5 мс
}
SetParent(hButton, hTaskBar);
```

Первые две строки нам уже знакомы. Здесь мы находим Панель задач и кнопку **Пуск**, сохраняя значения в глобальных переменных. Почему в глобальных, а не прямо в процедуре? Просто эти переменные мы будем использовать и дальше в этой программе (для других шуток). Поэтому, чтобы не выполнять один и тот же поиск, который всегда будет выдавать один и тот же результат, лучше сохраним один раз полученные значения в глобальной памяти программы.

В третьей строке загружается меню в переменную `MainMenu` с помощью функции `LoadMenu`. У этой функции два параметра — указатель на экземпляр и имя загружаемого меню. Это меню будет использоваться позже, а пока мы только подготовили переменную на будущее.

Теперь в этой функции будем подбрасывать кнопку **Пуск**. Но прежде чем это сделать, надо вспомнить, где она находится. Кнопка принадлежит Панели задач (расположена на ней), а значит, если мы начнем двигать что-то сейчас, то кнопка будет стоять на месте. Почему? Потому что кнопка не сможет оторваться от своей панели. Первым делом необходимо разорвать связь между

кнопкой и Панелью задач. Для этого выполняем функцию `SetParent` со следующими параметрами:

- окно, родителя которого нужно изменить, — наша кнопка;
- новый родитель указанного окна — указываем нуль.

Таким образом, у кнопки после выполнения этой функции будет "нулевой" родитель, и связь будет разрушена.

Вот теперь можно двигать кнопку как угодно. Поэтому следующий код будет вам знаком. Он практически не отличается от кода из листинга 2.3, где мы двигали окно, но в данном случае двигается кнопка, поэтому в функции `SetWindowPos` в качестве первого параметра используется указатель на кнопку, чтобы двигать именно ее.

После подъема и опускания кнопки **Пуск** мы должны вернуть ее на место, поэтому снова выполняем функцию `SetParent`, чтобы установить в качестве родителя для кнопки Панель задач.

На рис. 2.6 можно увидеть результат работы программы, который вы получите, если уже сейчас запустите пример. Обратите внимание, что на том месте, где должна быть кнопка **Пуск**, — пустота.

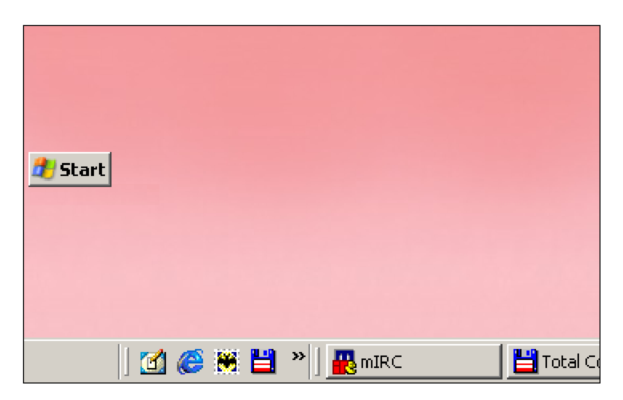


Рис. 2.6. Результат работы программы

Для реализации всех шуток в одной программе мы создадим несколько пунктов меню, с помощью которых будем вызывать разные команды. Для создания меню откройте ресурсы и выберите соответствующий пункт дерева (рис. 2.7).

Вы увидите свое меню, по которому можно двигаться. В конце каждого списка меню есть пустой пункт с именем **Type Here**, в котором имя написано на белом фоне. Выберите самый правый пункт с именем **Type Here** и введите новое имя **Our menu**. Название этого пункта и цвет изменятся, и таким способом мы получим новое меню, а справа от него, ниже, появится новый под-

пункт с именем **Type Here**. Таким образом, мы создаем новые пункты (подпункты) меню, которые будут видны в программе.

Выберите пункт **Our menu** и перетащите его мышью, чтобы он оказался перед пунктом **Help**, как на рис. 2.8.

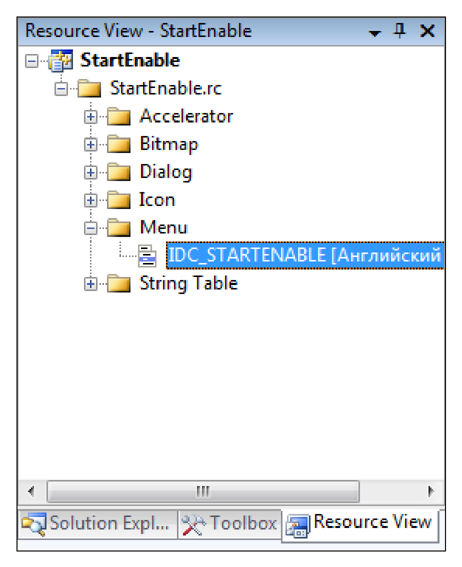


Рис. 2.7. Ресурсы и пункт дерева, под которым прячется меню

Рис. 2.8. В центре окна редактор меню

Теперь выделите пункт **Our menu**. Перейдите на пункт меню **Type Here**, который находится ниже, и наберите новое имя **Move window to System Tray**. Имя изменится, а строкой ниже появится новый пункт **Type Here**. Выделите его и создайте новый пункт **Enable System Tray**. Добавьте еще два пункта: **Disable System Tray** и **Insert menu**. В результате ваше меню должно выглядеть так, как на рис. 2.9.

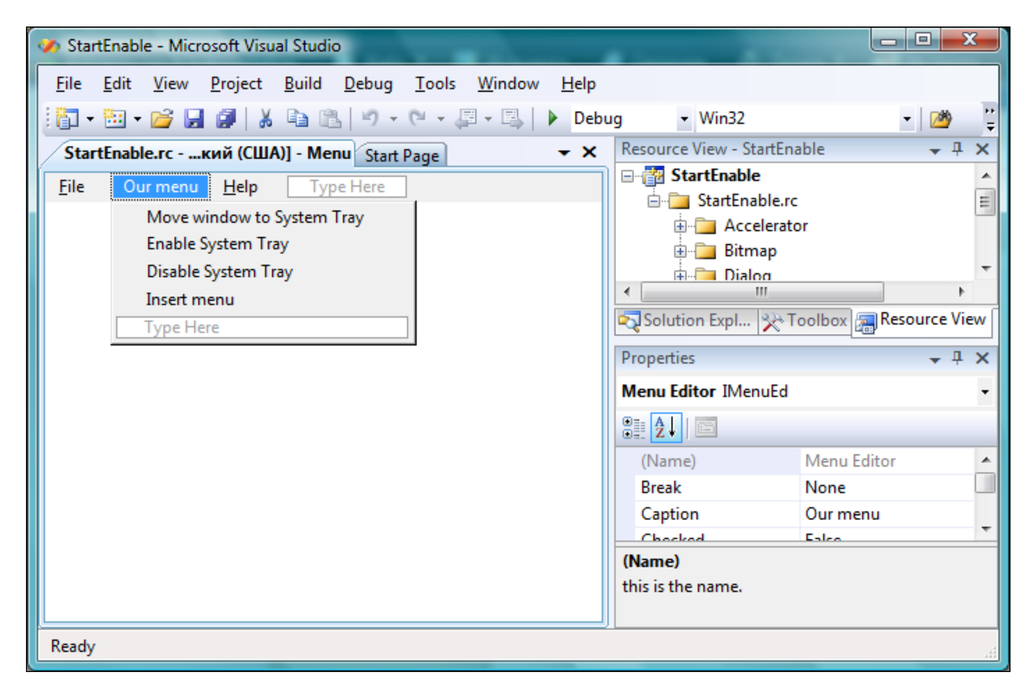


Рис. 2.9. Результат создания меню

Теперь приступаем к программированию. Перейдите в исходный код программы и найдите функцию `WndProc`. Когда пользователь выбирает какой-то пункт меню, то генерируется событие, и мы должны его обрабатывать в этой функции.

Полный код функции `WndProc` вы можете увидеть в листинге 2.8. Просмотрите его и приведите свою функцию к такому же виду.

Листинг 2.8. Функция обработки сообщений `WndProc`

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;
```

```
    switch (message)
    {
    case WM_COMMAND:
           wmId    = LOWORD(wParam);
           wmEvent = HIWORD(wParam);
           // Parse the menu selections:
           switch (wmId)
           {
           // Обрабатываем меню
           // Пункт меню Move window to System Tray
           case ID_OURMENU_MOVEWINDOWTOSYSTEMTRAY:
                  SetParent(hWnd, hTaskBar);
                  break;
           // Пункт меню Enable System Tray
           case ID_OURMENU_ENABLESYSTEMTRAY133:
                  EnableWindow(hTaskBar, true);
                  break;
           // Пункт меню Disable System Tray
           case ID_OURMENU_DISABLESYSTEMTRAY:
                  EnableWindow(hTaskBar, false);
           // Пункт меню Insert menu
                  break;
           case ID_OURMENU_INSERTMENU:
                  SetMenu(hTaskBar, MainMenu);
                  break;
           case IDM_ABOUT:
                  DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd,
                             LGPROC)About);
                  break;
           case IDM_EXIT:
                  DestroyWindow(hWnd);
                  break;
           default:
                 return DefWindowProc(hWnd, message, wParam, lParam);
           }
           break;
        ...
        // Здесь код удален для экономии места
        ...
    }
    return 0;
}
```

Большая часть из приведенного в листинге 2.8 кода была сгенерирована мастером при создании проекта. Нами добавлено не так уж и много, и сейчас нам предстоит это разобрать по частям.

Для первого пункта меню (**Move window to System Tray**) в данной функции выполняется следующий код:

```
// Пункт меню Move window to System Tray
case ID_OURMENU_MOVEWINDOWTOSYSTEMTRAY:
       SetParent(hWnd, hTaskBar);
break;
```

Оператор `case` проверяет пришедшее сообщение с константой `ID_OURMENU_MOVEWINDOWTOSYSTEMTRAY`. Если вы перейдете в редактор ресурсов и выберете созданный нами пункт меню **Move window to System Tray**, то в окне свойств (справа внизу) в свойстве `ID` увидите именно эту константу. Если она отличается, то для вас Visual C++ сгенерировал другое имя (у нас может отличаться версия среды разработки), и вы должны подкорректировать исходный код. Если проверка прошла успешно, то выполнится весь код до оператора `break`. Здесь идет вызов только одной функции `SetParent`. Мы уже знаем, что эта функция изменяет родителя окна, указанного в качестве первого параметра, делая его подчиненным по отношению к окну, заданному вторым параметром. Первым у нас указано главное окно, а вторым — Панель задач. Получается, что наше окно становится подчиненным по отношению к Панели задач.

На рис. 2.10 показан результат работы, который вы сможете получить, если выберете этот пункт меню. Я специально раздвинул Панель задач, чтобы вы видели, что окно **StartEnable** стало располагаться внутри этой панели. Вы больше не сможете выдвинуть окно программы за пределы Панели задач, пока не смените родителя на "нулевого".

При выборе пункта меню **Disable System Tray** выполняется следующий код:

```
// Пункт меню Disable System Tray
case ID_OURMENU_ENABLESYSTEMTRAY133:
    EnableWindow(hTaskBar, false);
    break;
```

Здесь мы выполняем функцию `EnableWindow`, которая делает доступным или недоступным какое-то окно. В качестве первого параметра мы передаем указатель на окно, а второй параметр равен `true` (сделать доступным) или `false` (сделать недоступным). В данном случае мы делаем недоступной Панель задач. Вы сколько угодно можете нажимать на кнопки панели, но ничего, кроме звукового сигнала об ошибке, не услышите. Но можно было бы указать и `hButton`, чтобы заблокировать только кнопку **Пуск**, а не всю Панель задач.

Рис. 2.10. Главное окно нашей программы стало подчиненным по отношению к Панели задач

Если выбрать пункт меню **Disable System Tray**, выполнится та же функция `EnableWndow`, только теперь мы сделаем окно доступным.

Выбор пункта меню **Insert menu** активизирует функцию `SetMenu`, которая устанавливает меню окна. Первым параметром определяется окно, а во втором параметре указывается загруженное меню. Вот и пригодилась переменная `MainMenu`, в которую мы в самом начале загрузили меню.

Посмотрите на рис. 2.11, там показан результат работы программы после выбора этого пункта меню. Самое интересное, что мышью вы его выбрать не можете. Единственный вариант войти в него — это клавиатура. Чтобы выбрать меню с помощью клавиатуры, нужно сделать активным окно (щелкните на Панели задач) и нажать клавишу <Alt>. Первый пункт меню должен подсветиться, и теперь вы можете перемещаться по пунктам меню с помощью клавиатуры и мыши.



Исходный код и запускаемый файл этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter2\StartEnable.

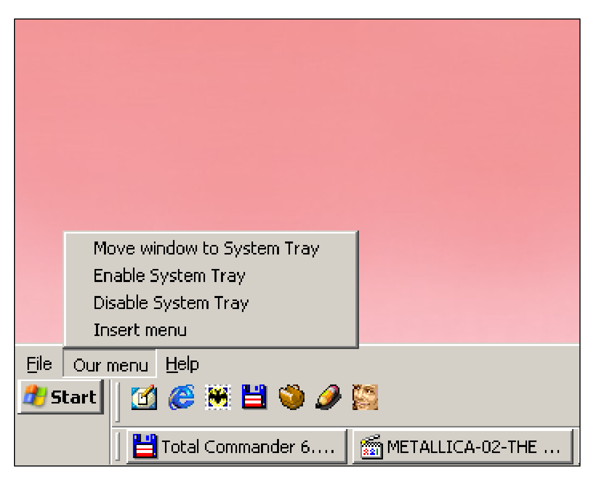


Рис. 2.11. Меню в Панели задач — нонсенс или реальность?

2.5. Маленькие шутки

Рассмотрим несколько маленьких приколов. Это небольшие задачи, ради которых нет смысла писать самостоятельные примеры, поэтому в целях экономии места я объединил различные шутки в одну программу. Вы можете использовать эту заготовку в своих невидимых шуточных приложениях или реальных программах. Некоторые используемые функции могут пригодиться и в коммерческих проектах.

2.5.1. Как программно потушить монитор

Не знаю, как программно, а огнетушителем тушится за пять секунд! Я даже помню, как в детстве получил значок юного огнетушителя (в смысле — пожарника).

А если серьезно, то системная команда "на тушение" выглядит так:

```
SendMessage(hWnd, WM_SYSCOMMAND, SC_MONITORPOWER, 0);
```

Чтобы "зажечь", измените значение последнего параметра на `-1`.

2.5.2. Запуск системных CPL-файлов

Добавьте в начало файла модуль `shellapi.h`, чтобы использовать функцию `ShellExecute`:

```
#include <shellapi.h>
```

Теперь напишите следующий код:

```
ShellExecute(hWnd, "Open", "Rundll32.exe",
 "shell32,Control_RunDLL filename.cpl", "",SW_SHOWNORMAL);
```

Функция `ShellExecute` запускает указанную программу. У нее есть следующие параметры:

- окно, из которого запускается программа, — можно указать хоть 0, для нас это не важно;
- действие, которое надо выполнить, — для запуска программы указываем `"Open"`;
- имя запускаемой программы;
- команды, которые надо передать в командной строке;
- каталог по умолчанию, из которого будет работать запущенная программа, — при задании пустой строки будет использоваться путь по умолчанию, что нас вполне устраивает;
- тип запуска — параметр, который указывает, как запустить программу, указываем `SW_SHOWNORMAL`, что означает запуск программы в нормальном режиме (флаг идентичен параметру функции `ShowWindow`).

Например, нам нужно запустить Rundll32.exe (умеет выполнять DLL- и CPL-файлы). В качестве команды нужно передать текст вот такого вида: `shell32,Control_RunDLL filename.cpl`.

Тогда вот такой код отобразит окно настроек Интернета:

```
ShellExecute(hWnd, "Open", "Rundll32.exe",
 "shell32,Control_RunDLL inetcpl.cpl", "",SW_SHOWNORMAL);
```

А такой код отобразит окно настроек экрана:

```
ShellExecute(hWnd, "Open", "Rundll32.exe",
 "shell32,Control_RunDLL desk.cpl", "",SW_SHOWNORMAL);
```

2.5.3. Программное управление CD-ROM

Очень хорошая шутка — открытие и закрытие лотка CD-ROM. Вы можете организовать цикл и бесконечно открывать и закрывать дверцу. Мы же рассмотрим пример единичного открытия.

Итак, нужно добавить заголовочный файл mmsystem.h. Это можно сделать в начале файла или в заголовочном файле stdafx.h следующим образом:

```
#include <mmsystem.h>
```

Теперь в окне **Solution Explorer** переместите указатель на строку с именем вашего проекта и выберите меню **Project/Properties**. В открывшемся окне (рис. 2.12) в дереве слева установите **Configuration Properties/Linker/Input/**. Функции, которые мы сейчас будем использовать, расположены в библиотеке winmm.lib, а она при сборке проекта по умолчанию не подключается к запускаемому файлу. Поэтому мы должны подключить эту библиотеку вручную. Для этого в поле **Additional Dependencies** напишите имя библиотеки winmm.lib.

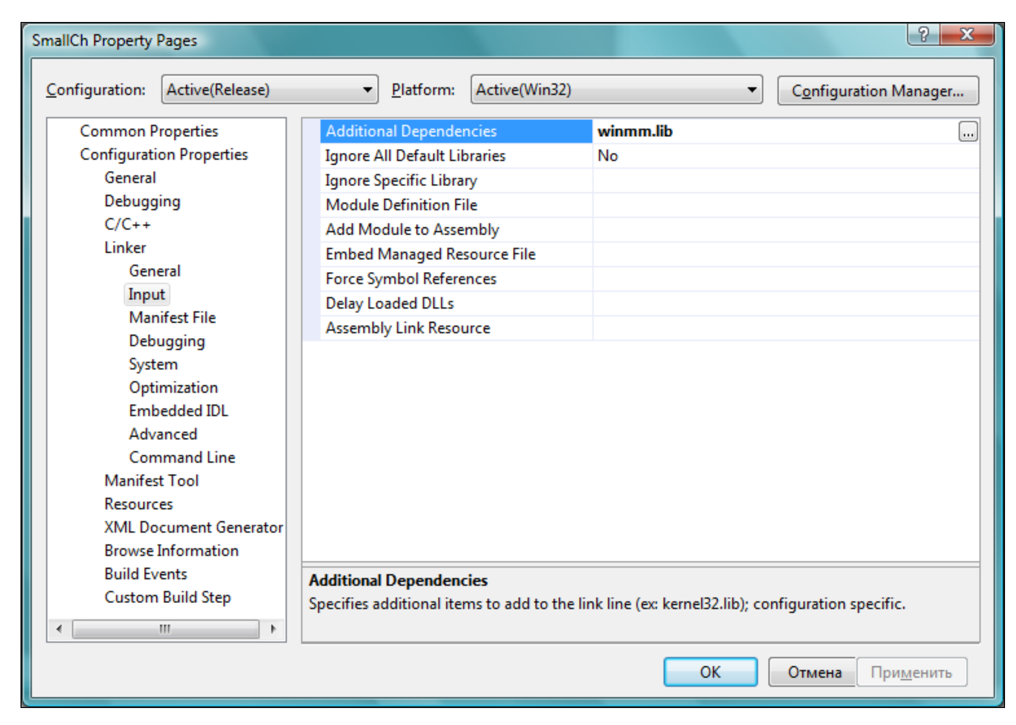


Рис. 2.12. Настройки командной строки сборщика проекта

Для работы нам понадобятся следующие переменные:

```
MCI_OPEN_PARMS OpenParm;
MCI_SET_PARMS SetParm;
MCIDEVICEID dID;
```

Сам код открытия и закрытия CD-ROM будет выглядеть следующим образом:

```
OpenParm.lpstrDeviceType="CDAudio";
mciSendCommand(0, MCI_OPEN, MCI_OPEN_TYPE, (DWORD_PTR)&OpenParm);
dID = OpenParm.wDeviceID;
```

```
mciSendCommand(dID, MCI_SET, MCI_SET_DOOR_OPEN,(DWORD_PTR)&SetParm);
mciSendCommand(dID,MCI_SET,MCI_SET_DOOR_CLOSED,(DWORD_PTR)&SetParm);
mciSendCommand(dID, MCI_CLOSE, MCI_NOTIFY, (DWORD_PTR)&SetParm);
```

Сначала мы должны определить параметр `lpstrDeviceType` структуры `OpenParm`. Ему нужно присвоить значение строки `"CDAudio"`, что и будет указывать на необходимость работы с CD-ROM.

Для работы с мультимедийными устройствами, к которым относится и CD-ROM, используется функция `mciSendCommand`. Она отправляет устройству сообщение и передает следующие параметры:

- идентификатор устройства, которое должно получить сообщение, — значение получаем при открытии устройства, поэтому если в качестве второго параметра указан флаг `MCI_OPEN`, то параметр игнорируется, т. к. устройство еще не открыто;
- команда сообщения;
- флаг для сообщения, которое должно быть послано устройству;
- указатель на структуру, которая содержит параметры для команды сообщения.

В первый раз мы посылаем сообщение `MCI_OPEN`, чтобы открыть устройство. После этого в параметре `wDeviceID` структуры `OpenParm` будет находиться идентификатор открытого устройства. Именно его мы будем использовать в качестве первого параметра для отправки сообщений.

Чтобы открыть дверцу CD-ROM, нужно отправить сообщение, в котором второй параметр равен `MCI_SET`, а третий — `MSI_SET_DOOR_OPEN`. Последний параметр нас не интересует. Закрытие дверцы похоже на открытие, только третий параметр равен `MSI_SET_DOOR_CLOSED`.

После завершения работы с устройством мы должны его закрыть. Для этого отправляем сообщение, в котором второй параметр равен `MCI_CLOSE`, а третий — `MCI_NOTIFY`.

2.5.4. Удаление часов из Панели задач

Удаление выполняется почти так же, как с кнопкой **Пуск**. Нужно сначала найти окно Панели задач. Потом на нем найти окно **TrayBar**, на котором уже найти часы. После этого часики легко убираются функцией `ShowWindow`, которой нужно передать в качестве первого параметра указатель на окно часов, а во втором параметре указать `SW_HIDE`.

```
HWND Wnd;
Wnd = FindWindow("Shell_TrayWnd", NULL);
```

```
Wnd = FindWindowEx(Wnd, HWND(0),"TrayNotifyWnd", NULL);
Wnd = FindWindowEx(Wnd, HWND(0),"TrayClockWClass", NULL);
ShowWindow(Wnd, SW_HIDE);
```

Можно было бы спрятать всю панель с пиктограммами в правом углу Панели задач. Для этого достаточно не использовать строку кода с параметром `"TrayClockWClass"`.

2.5.5. Исчезновение чужой программы

Как работать с чужими окнами, мы еще подробно рассмотрим в следующих главах книги. Но все же я приведу вам один интересный пример с исчезновением чужих программ:

```
HWND Wnd;
while (true)
{
        Wnd=GetForegroundWindow();
        if (Wnd>0)
               ShowWindow(Wnd,SW_HIDE);
        Sleep(1000);
};
```

В этом примере запускается бесконечный цикл `while`, внутри которого выполняются следующие шаги:

1. Получаем идентификатор активного окна с помощью функции `GetForegroundWindow`.
2. Прячем окно с помощью функции `ShowWindow`, если идентификатор "правильный" (больше нуля).
3. Делаем задержку в 1 секунду на реакцию пользователя.

Если выполнить этот код, то любое активное окно исчезнет максимум через одну секунду. Даже если попытаться снять задачу, которая выполняет этот код, то за одну секунду вы не успеете вызвать Панель задач, найти программу и снять ее, т. к. уже исчезнет Диспетчер задач. Невозможно нажать на кнопку **Пуск** и завершить работу, потому что исчезнет сама Панель задач, которая в данный момент станет активной.

Именно поэтому перед запуском примера я настоятельно рекомендую сохранить все открытые документы, чтобы ничего не пропало. Кроме того, необходимо предусмотреть возможность отключения цикла.

2.5.6. Установка на Рабочий стол собственных обоев

Задача проще некуда:

```
SystemParametersInfo(SPI_SETDESKWALLPAPER, 0, "c:\\1.bmp",
     SPIF_UPDATEINIFILE);
```

Функция `SystemParametersInfo` имеет следующие параметры:

- действие, которое надо выполнить, — этих действий очень много и описывать все излишне, привожу самые интересные:
  - `SPI_SETDESKWALLPAPER` — установить собственные обои. Путь к файлу с обоями должен быть передан в третьем параметре;
  - `SPI_SETDOUBLECLICKTIME` — время двойного щелчка. Количество миллисекунд между первым и вторым щелчком мыши нужно указать во втором параметре. Попробуйте указать здесь число меньше 10, и я думаю, что вы никогда не успеете за это время "кликнуть" дважды. Таким образом, практически отключается функция двойного щелчка;
  - `SPI_SETKEYBOARDDELAY` — во втором параметре устанавливается задержка между нажатиями клавиш на клавиатуре при удерживании кнопки;
  - `SPI_SETMOUSEBUTTONSWAP` — если во втором параметре 0, то кнопки мыши используются стандартно, иначе кнопки меняются местами, как для левши;
- второй параметр зависит от состояния первого;
- третий параметр зависит от состояния первого;
- четвертым параметром устанавливаются флаги, в которых указано, что надо делать после выполнения действия. Возможны следующие варианты:
  - `SPIF_UPDATEINIFILE` — обновить пользовательский профиль;
  - `SPIF_SENDCHANGE` — сгенерировать `WM_SETTINGCHANGE`-сообщение;
  - `SPIF_SENDWININICHANGE` — то же, что и предыдущий параметр.

Если функция выполнилась удачно, то она вернет любое число, не равное нулю, иначе функция вернет ноль. Пример кода, который меняет кнопки мыши местами:

```
// Установить мышь для левши
SystemParametersInfo(SPI_SETMOUSEBUTTONSWAP, 1, 0,
 SPIF_SENDWININICHANGE);
```

```
// Вернуть на родину
SystemParametersInfo(SPI_SETMOUSEBUTTONSWAP, 0, 0,
 SPIF_SENDWININICHANGE);
```



Все примеры, описанные в этом разделе, вы можете найти на компакт-диске в папке \Demo\Chapter2\SmallCh.

2.6. Шутки с мышкой

Мышка тоже может быть объектом насмешек. Например, можно заставить ее беспорядочно бегать по монитору или ограничить движение маленьким квадратом. А можно вообще остановить указатель в определенной точке, создав видимость его зависания. Как показывает практика, игры с мышкой производят на пользователя большее впечатление, особенно на начинающего, потому что он больше работает с ней, а не с клавиатурой.

2.6.1. Безумная мышка

Как сделать мышку безумной? Очень даже просто:

```
for (int i=0; i<20; i++)
{
       SetCursorPos(rand()%640, rand()%480);
       Sleep(100);
}
```

Здесь запускается цикл от 0 до 20, в котором указатель мыши переносится в случайную позицию с помощью функции `SetCursorPos`, которой нужно передать два параметра `X` и `Y` в виде целых чисел — координаты новой позиции курсора. Параметры формируются с помощью функции `rand()` в диапазоне от нуля до числа, указанного после знака %. В реальной программе желательно определять разрешение экрана (ширину и высоту) с помощью функции `GetSystemMetrics` с параметрами `SM_CYSCREEN` и `SM_CXSCREEN`, но в примере для наглядности я ограничился разрешением 640×480.

После каждого перемещения делается задержка в 20 с, чтобы пользователь успел заметить указатель в новом положении. Таким образом, мышка совершит 20 прыжков по экрану.

2.6.2. Летающие объекты

В *разд. 2.2* мы рассматривали пример, в котором программно щелкали по кнопке **Пуск**. Тогда это было определенное окно, поэтому задача упроща-

лась. Поставим задачу шире — щелкнуть в любой области экрана. Для этого потребуется определить, какое в этом месте окно. Этот вопрос тоже решается достаточно просто:

```
for (int i=0; i<20; i++)
{
	POINT pt = {rand()%800, rand()%600};
	SetCursorPos(pt.x, pt.y);
	Sleep(100);

	HWND hPointWnd = WindowFromPoint(pt);
	SendMessage(hPointWnd, WM_LBUTTONDOWN, MK_LBUTTON,
	  MAKELONG(pt.x,pt.y));
	SendMessage(hPointWnd, WM_LBUTTONUP, 0,
	  MAKELONG(pt.x,pt.y));
}
```

В этом примере, как и в задаче с мышкой, мы случайным образом генерируем две координаты и сохраняем их в параметрах `x` и `y` структуры `pt`. Потом изменяем положение курсора в соответствии с полученными координатами.

На следующем шаге определяем окно, которое находится в этой позиции. Для этого существует функция `WindowFromPoint`, которой нужно передать один параметр — структуру типа `POINT` с хранящимися в ней координатами искомой точки. Функция вернет указатель на это окно.

Теперь отправляем два сообщения уже знакомым способом. В первом случае второй параметр равен `WM_LBUTTONDOWN` (когда нажата левая кнопка мыши), а во втором — `WM_LBUTTONUP` (когда отпущена). Почему здесь два события? Если кнопка **Пуск** реагирует на нажатия, то программы чаще всего обрабатывают полное нажатие (`Click`) или событие, когда отпущена кнопка. Поэтому желательно отправлять оба события.

В качестве последнего параметра функции `SendMessage` мы передаем координаты точки, где щелкнул пользователь. Для этого обе координаты собираются в одно большое число с помощью макроса `MAKELONG` (макрос похож по своему принципу работы на функцию).

Можно сразу немного изменить пример:

```
for (int i=0; i<20; i++)
{
	// Устанавливаем случайную позицию курсора
	POINT pt = {rand()%800, rand()%600};
	SetCursorPos(pt.x, pt.y);
	Sleep(100);
```

```
        // Посылаем сообщение о нажатии кнопки мыши
        HWND hPointWnd = WindowFromPoint(pt);
        SendMessage(hPointWnd, WM_LBUTTONDOWN, MK_LBUTTON,
          MAKELONG(pt.x,pt.y));

        // Изменение позиции курсора
        POINT pt1 = {rand()%800, rand()%600};
        SetCursorPos(pt1.x, pt1.y);

        SendMessage(hPointWnd, WM_MOUSEMOVE, 0,
          MAKELONG(pt1.x,pt1.y));

        // Отпускаем кнопку мыши
        SendMessage(hPointWnd, WM_LBUTTONUP, 0,
          MAKELONG(pt1.x,pt1.y));
}
```

Здесь между событиями нажатия и отпускания кнопки мы генерируем новые координаты, перемещаем туда мышь и отпускаем ее. Перед тем как послать сообщение о том, что кнопка отпущена, мы отправляем сообщение `WM_MOUSEMOVE`, которое заставляет программу отработать перемещение указателя мыши. Таким образом, нажали в одном месте, а отпустили в другом. И если в месте нажатия был объект, который можно перетащить, то он перелетит в новое место (получается программный Drag&Drop).

2.6.3. Мышка в клетке

Очень интересным примером является ограничение свободы перемещения мышки. Посмотрите на следующий код:

```
RECT r;
r.left=10;
r.top=10;
r.bottom=100;
r.right=100;
ClipCursor(&r);
```

Определим переменную `r` типа `RECT`. Это структура, которая состоит из четырех числовых переменных, описывающих прямоугольник. Переменные структуры имеют следующие имена: `left`, `top`, `bottom` и `right` (левая, верхняя, нижняя и правая координаты прямоугольника).

В следующих строках мы присваиваем этим переменным значения, определяя тем самым прямоугольную область. Затем вызываем функцию `ClipCursor`,

которая и ограничивает движение курсора мышки указанным прямоугольником.

Попробуйте выполнить следующий код:

```
RECT r;
r.left=0;
r.top=0;
r.bottom=1;
r.right=1;
ClipCursor(&r);
```

Здесь размер области передвижения равен 1 пикселу по горизонтали и вертикали, поэтому мышка окажется запертой в клетке.

2.6.4. Изменчивый указатель

Есть такая интересная WinAPI-функция — `SetSystemCursor`. У нее есть два параметра:

- курсор, который надо изменить. Чтобы восстановить системный курсор, можно использовать функцию `GetCursor`;
- вид курсора, который нужно установить. Здесь можно указать одно из следующих значений:
  - `OCR_NORMAL` — стандартный курсор (по умолчанию);
  - `OCR_IBEAM` — курсор, используемый для выделения текста;
  - `OCR_WAIT` — большие песочные часы (ожидание);
  - `OCR_CROSS` — графическое выделение (крест);
  - `OCR_UP` — стрелка вверх;
  - `OCR_SIZE` — курсор изменения размера;
  - `OCR_ICON` — значок;
  - `OCR_SIZENWSE` или `OCR_SIZENESW` — курсор, используемый для растягивания объекта;
  - `OCR_SIZEWE` — курсор для горизонтального изменения размера;
  - `OCR_SIZENS` — курсор для вертикального изменения размера;
  - `OCR_SIZEALL` — курсор для одновременного изменения размера по горизонтали и вертикали;
  - `OCR_SIZENO` — интернациональный несимвольный курсор;
  - `OCR_APPSTARTING` — маленькие песочные часы (загрузка приложения).

И сразу приведу небольшой пример изменения текущего курсора:

```
SetSystemCursor(GetCursor(), OCR_CROSS);
```

Этот код изменяет текущий курсор на крестик, который используется при графическом выделении.

2.6.5. Скоростной режим

При движении по дороге нужно соблюдать скоростной режим, иначе можно попасть в аварию. При движении по монитору в аварию не попадешь, разве что с краем экрана, но скоростной режим указателя также можно ограничить. Следующий пример показывает, как изменить скорость движения мыши:

```
int speed=1;
SystemParametersInfo(SPI_SETMOUSESPEED, NULL,
    (void*)speed, SPIF_UPDATEINIFILE);
```



Все примеры, описанные в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter2\JokesWinMouse.

2.7. Найти и уничтожить

Попробуем найти определенное окно и уничтожить его. Для этого создайте новое приложение Win32 Project, а также пункт меню, по нажатию которого будет выполняться наш код.

Теперь перейдите в функцию `WndProc`, в которой обрабатываются все события окна. В начало этой функции нужно добавить переменную `h` типа `HWND`, что будет выглядеть примерно так:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
       int wmId, wmEvent;
       PAINTSTRUCT ps;
       HDC hdc;
       HWND h;
```

А дальше добавим обработчик события нашего меню:

```
case ID_MYCOMMAND_FINDANDDESTROY:
       h = FindWindow(0, "1 — Блокнот");
       if (h != 0)
              SendMessage(h, WM_DESTROY, 0,0);
       break;
```

С помощью уже знакомой нам функции `FindWindow` мы ищем окно, у которого в заголовке есть текст "1 — Блокнот". Результат поиска сохраняем в переменной `h`. В качестве параметра поиска задаем только заголовок окна, а класс окна оставляем пустым (его не всегда можно определить). В этом и состоит тонкость программы, потому что большинство приложений с открытым документом имеет заголовок типа "Имя документа — Название приложения", например, "MyTestDoc — Microsoft Word". Если открытых документов нет, в заголовке остается название программы — "Microsoft Word". Только в этом случае функция `FindWindow` может однозначно определить окно, которое надо найти, иначе она возвратит 0. Если проверка переменной `h` показывает, что окно найдено, то посылается сообщение `WM_DESTROY`, чтобы уничтожить его.

Как видите, поиск окна по заголовку нельзя назвать точным и надежным.

В этом примере окно уничтожается только по выбору соответствующего пункта меню.

А что, если перенести этот код в цикл обработки сообщений следующим образом:

```
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
	h=FindWindow(0, "1 — Блокнот");
	if (h!=0)
		SendMessage(h, WM_DESTROY, 0,0);

	if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
}
```

В этом случае при наступлении любого события в искомом окне программа найдет и уничтожит его. Можно пойти еще дальше и поместить поиск в бесконечный цикл, тогда программа, не дожидаясь какого-либо события, будет постоянно искать и уничтожать окно. Таким способом вы можете ограничить пользователя в запуске определенных программ.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter2\FindAndDestroy.

2.8. Блокировка Рабочего стола

Работа с Windows начинается с Рабочего стола, а это тоже окно со всеми вытекающими отсюда последствиями. Чтобы получить его указатель, надо воспользоваться функцией `GetDesktopWindow`. Рассмотрим несколько примеров, с помощью которых можно пошутить, используя Рабочий стол.

```
HWND h = GetDesktopWindow();
EnableWindow(h, FALSE);
```

В первой строке кода мы получаем указатель на окно, а во второй — делаем его неактивным. Попробуйте выполнить этот код в своей программе, и вы заблокируете Windows. Жаль, что блокировка не полная, и с помощью нажатия клавиш <Ctrl>+<Alt>+<Del> откроется Диспетчер задач, после чего блокировка исчезнет. Но если поместить этот код в бесконечный цикл или в цикл обработки сообщений, то Windows исчезнет "навсегда".



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter2\DesktopWindow.

2.9. Сетевая бомба

В Windows NT-системах (NT/2000/XP/2003) имеется очень интересная и удобная команда `NET SEND`, которая позволяет отправить на другой компьютер сообщение из командной строки. Вы пишете команду, адрес получателя и текст сообщения. А после выполнения инструкции на компьютере адресата появляется окно с текстом сообщения. Пример такого окна вы можете увидеть на рис. 2.13.

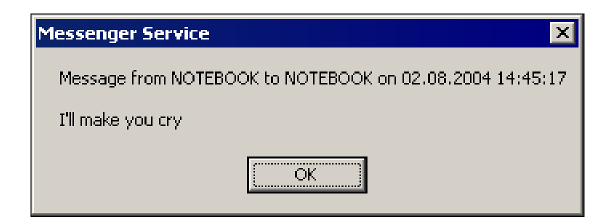


Рис. 2.13. Сообщение, посланное командой `NET SEND`

Сообщение отправляется командой следующего вида:

```
NET SEND Адрес Текст
```

В качестве адреса можно указывать как NETBios-имя компьютера, так и IP-адрес. Вот пример, который посылает сообщение "Hi, Dany" на компьютер Dany:

```
NET SEND Dany Hi, Dany
```

Самое интересное, что Windows 2000 и Windows XP абсолютно не защищены от бомбардировки командой `NET SEND`. Вы можете очень быстро послать хоть сто команд на компьютер вашего друга с любыми сообщениями, и все они дойдут. Но отправлять руками — утомительно, поэтому напишем небольшую программу.

Создайте новый проект типа Win32 Project в Visual C++, и в функции `_tWinMain` напишите следующий код до цикла обработки сообщений:

```
for (int i=0; i<10; i++)
{
WinExec("NET SEND 192.168.1.121 You will be cry by me",
SW_SHOW);
       Sleep(1000);
}
```

Здесь мы запускаем цикл, в котором функция `WinExec` будет 10 раз выполнять код, указанный в качестве первого параметра в командной строке Windows. В данном примере это текст "NET SEND 192.168.1.121 You will be cry by me". Если выполнить этот код в командной строке, то вы отправите сообщение "You will be cry by me" на компьютер с адресом 192.168.1.121.

На каждом шаге цикла мы делаем задержку в 1 секунду с помощью функции `Sleep`, чтобы между сообщениями была хоть какая-то пауза.

Если вас начали бомбить сообщениями `NET SEND`, то даже не пытайтесь успеть закрыть все окна. Выполните следующие действия:

1. Выдерните сетевой кабель, который связывает вас с сетью, из которой идет бомбардировка. Если это для вас неприемлемо, и связь нельзя выключать, то в любом случае переходите к следующему пункту.
2. Выполните последовательно **Пуск/Настройка/Панель управления/Администрирование/Службы** и в появившемся окне найдите строку "Служба сообщений" (рис. 2.14). Щелкните по ней правой кнопкой мыши и в появившемся меню выберите пункт **Стоп**.

Если вы не пользуетесь сообщениями, то лучше эту службу отключить заранее (по умолчанию включена), чтобы не было даже потенциальной возможности такой атаки на ваш компьютер.

Services

File Action View Help

Services (Local)

Name	Description	Status	Startup Type	Log On As
Kerberos Key Distribution Center	On domain ...		Disabled	Local System
License Logging	Monitors a...		Disabled	Network S...
Logical Disk Manager	Detects an...	Started	Automatic	Local System
Logical Disk Manager Administrative Service	Configures...		Manual	Local System
Messenger	Transmits ...	Started	Automatic	Local System
Microsoft Search	Creates ful...		Manual	Local System
Microsoft Software Shadow Copy Provider	Manages s...		Manual	Local System
MSSQLSERVER			Manual	Local System
MSSQLServerADHelper			Manual	Local System
Net Logon	Maintains a...		Manual	Local System
NetMeeting Remote Desktop Sharing	Enables an...		Disabled	Local System
Network Connections	Manages o...	Started	Manual	Local System
Network DDE	Provides n...		Disabled	Local System
Network DDE DSDM	Manages D...		Disabled	Local System
Network Location Awareness (NLA)	Collects an...	Started	Manual	Local System
NT LM Security Support Provider	Provides s...		Manual	Local System
Performance Logs and Alerts	Collects pe...		Manual	Network S...
Plug and Play	Enables a c...	Started	Automatic	Local System
Portable Media Serial Number Service	Retrieves t...		Manual	Local System
Print Spooler	Manages al...	Started	Automatic	Local System
Protected Storage	Protects st...	Started	Automatic	Local System
Remote Access Auto Connection Manager	Detects un...		Manual	Local System
Remote Access Connection Manager	Manages di...	Started	Manual	Local System
Remote Desktop Help Session Manager	Manages a...		Manual	Local System
Remote Procedure Call (RPC)	Serves as t...	Started	Automatic	Local System

Extended Standard

Рис. 2.14. Окно управления службами



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter2\NetBomb.

ГЛАВА 3



Система

В этой главе будут рассматриваться разные системные утилиты. Сюда войдут примеры программ, способных следить за происходящим в системе. Это уже не просто программы-приколы, а работа с системой, хотя шуток в рассматриваемых задачах будет достаточно. Как я уже говорил, любой хакер — это профессионал, а значит, должен знать и уметь работать с "внутренностями" той ОС, в которой он находится.

Слишком низко мы опускаться не будем, поэтому до уровня ядра мы не упадем, чтобы книга была интересна не только опытным, но и начинающим. Дело в том, что для спуска на нижний уровень нужно использовать неофициальные методы или писать драйверы. Первое проблематично, потому что разработчик постоянно латает свои ошибки и запрещает вызовы привилегированных функций. К моменту выхода книги те методы, которые я знаю, уже будут закрыты. Второе проблематично, потому что требует отдельного класса знаний, и о драйверах пишут целые книги. Поэтому мы будем рассматривать официальные методы.

При создании книги я подразумевал, что вы находитесь в ОС Windows, программируете и работаете в ней. В данной главе я попробую научить вас лучше понимать эту систему. Я постараюсь не загружать вас теорией, а дать максимум практики. Если вы уже читали мои труды, то знаете мой стиль. Я всегда говорю, что только практика ведет к познанию. Грош цена тем знаниям, которые не применить на практике. Именно поэтому все главы данной книги наполнены практическими примерами, и эта — не исключение.

Я покажу несколько интересных примеров, и мы подробно разберем их. Таким образом, мы рассмотрим некоторые особенности работы с ОС Windows, и вы поймете, как применять эти особенности на практике. Надеюсь, что это вам поможет в работе.

В этой главе я постепенно буду усложнять примеры и покажу много интересного и полезного.

3.1. Работа с чужыми окнами

Я регулярно получаю письма с вопросами типа: "Как уничтожить чужое окно или изменить что-то в нем?". В принципе, эта задача легко решается с помощью уже знакомой нам функции `FindWindow`. Но если необходимо изменить множество окон (или даже все), то нужно использовать другой метод поиска, который мы сейчас рассмотрим. Для начала напишем программу, которая будет искать все окна на Рабочем столе и изменять их заголовки.

На рис. 3.1 показан вид нескольких окон после запуска программы, которую нам сейчас предстоит написать. Как видите, все заголовки изменились на "I See You".

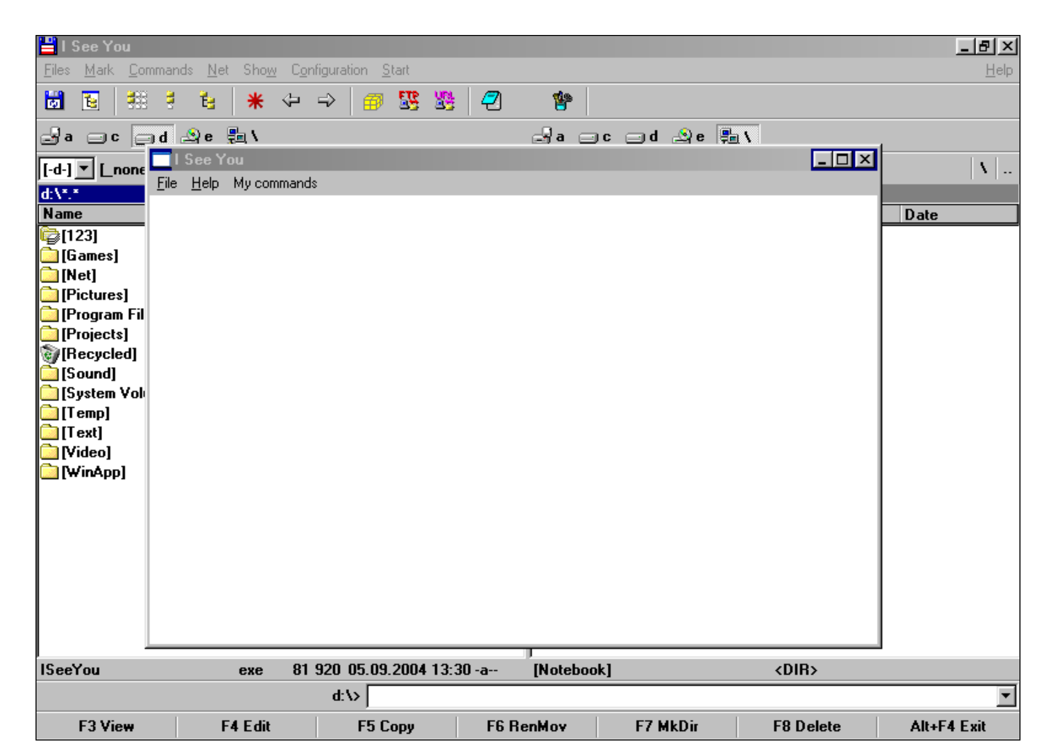


Рис. 3.1. Результат работы программы ISeeYou

Создайте в Visual C++ новый проект Win32 Project и в нем какой-нибудь пункт меню, при выборе которого будет запускаться программа, реализующая нашу задачу.

В функции `WndProc` добавьте следующий код обработки пункта меню:

```
case ID_MYCOMMANDS_ISEEYOU:
       while (TRUE)
       {
              EnumWindows(&EnumWindowsWnd, 0);
       }
```

В приведенном коде `ID_MYCOMMANDS_ISEEYOU` — это идентификатор пункта меню.

Цикл `while` будет выполняться бесконечно (`TRUE` никогда не станет равным `FALSE`). Внутри цикла вызывается функция `EnumWindows`. Это WinAPI-функция, которая используется для перечисления всех открытых окон. В качестве первого параметра ей нужно передать адрес другой функции, которая будет вызываться каждый раз, когда найдено какое-нибудь запущенное окно. В качестве второго параметра указывается число, которое будет передаваться в функцию обратного вызова.

В качестве функции обратного вызова будет использоваться функция `EnumWindowsWnd`. Так что каждый раз, когда `EnumWindows` найдет окно, будет выполняться код, написанный в `EnumWindowsWnd`. Этот код выглядит следующим образом:

```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,     // handle to parent window
                   // (Указатель на главное окно)
    LPARAM lParam  // application-defined value
                   // (значение, определенное приложением)
   )
{
       SendMessage(hwnd, WM_SETTEXT, 0,LPARAM(LPCTSTR("I See You")));
       return TRUE;
}
```

Количество параметров, их тип и тип возвращаемого значения должны быть именно такими:

- идентификатор найденного окна типа `HWND`;
- значение типа `LPARAM`, которое вы можете использовать в своих целях.

Если что-то изменить, то функция станет несовместимой с `EnumWindows`. В таких случаях, чтобы не ошибиться, я беру имя функции с параметрами прямо из файла помощи по WinAPI и вставляю в свой код. Лучше лишний раз проверить, чем потом долго искать опечатку или случайную ошибку. То же самое я советую делать и вам. В данном случае нужно открыть файл по-

мощи в разделе EnumWindows и перейти по ссылке, указывающей на формат функции обратного вызова.

Итак, у нас есть идентификатор найденного окна. Такой параметр мы уже использовали много раз, когда прятали или перемещали окно, теперь научимся изменять его заголовок. Для этого используем уже знакомую функцию SendMessage, которая посылает сообщения Windows. Вот ее параметры:

- ❒ идентификатор окна, которому надо отослать сообщение — передан в качестве параметра функции-ловушки EnumWindowsWnd, и он равен идентификатору найденного окна;
- ❒ тип сообщения — WM_SETTEXT, заставляет окно сменить текст заголовка;
- ❒ параметр для данного сообщения должен быть 0;
- ❒ новое имя окна.

Чтобы программа продолжила поиск следующего окна, ей надо вернуть значение TRUE.



Исходный код этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter3\ISeeYou.

Давайте немного усложним пример. Для начала изменим функцию EnumWindowsWnd до следующего вида:

```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,    // handle to parent window
    LPARAM lParam // application-defined value
   )
{
       SendMessage(hwnd, WM_SETTEXT, 0,LPARAM(LPCTSTR("I See You")));
       EnumChildWindows(hwnd,&EnumChildWnd,0);
       return TRUE;
}
```

Здесь после отправки сообщения вызывается функция EnumChildWindows, которая определяет все окна, принадлежащие главному окну. У нее три параметра:

- ❒ идентификатор окна, дочерние элементы которого нужно искать, — указываем окно, которое уже нашли;
- ❒ адрес функции обратного вызова, которая будет запускаться каждый раз, когда найдено дочернее окно;
- ❒ просто число, которое может быть передано в функцию обратного вызова.

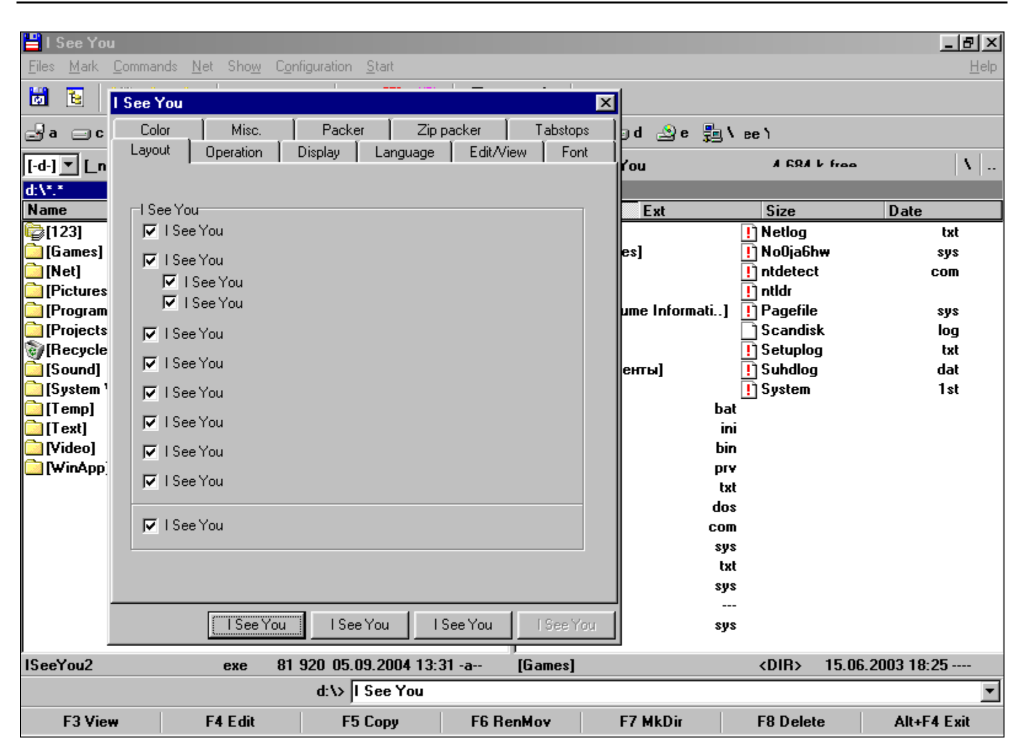


Рис. 3.2. Результат работы программы ISeeYou2

Как вы можете заметить, работа функции `EnumChildWnd` похожа на `EnumWindowsWnd`, но если вторая ищет окна среди запущенных программ, то первая — внутри указанного окна. Образец такого окна можно увидеть на рис. 3.2.

В этой функции также изменяется заголовок найденного окна, а чтобы поиск продолжился дальше, функция возвращает `TRUE`.

В принципе программу можно считать законченной, но у нее есть один недостаток, о котором нельзя умолчать. Допустим, что программа нашла окно и начала перечисление в нем дочерних окон, а в этот момент окно закрывают. Программа пытается послать сообщение найденному окну об изменении текста, а его уже не существует, и происходит ошибка выполнения. Чтобы этого избежать, в самом начале функций обратного вызова нужно поставить проверку на правильность полученного идентификатора окна:

```
if (h==0)
      return TRUE;
```

Вот теперь приложение можно считать завершенным, абсолютно рабочим и более-менее стабильным. Помните, что лишних проверок не бывает. Если

хотите, чтобы ваш код был надежным, то нужно проверять все, что может вызвать проблемы. В данной книге я иногда буду пренебрегать этим правилом, чтобы не усложнять и не запутывать программу, но постараюсь указывать на те участки кода, которые нуждаются в пристальном внимании.

Примечание

Исходный код этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter3\ISeeYou2.

В примерах *главы 2* я старался писать весь код в цикле обработки сообщений. Таким образом, программа сначала выполняла предписанные действия, а потом обрабатывала сообщения системы. Для завершения такого приложения программу было достаточно просто закрыть. В данном же случае запускается бесконечный цикл, который выполняется вне обработчика сообщений системы. Таким образом, наш цикл будет постоянно работать, а сообщения системы обрабатываться не будут, т. к. на них не хватит процессорного времени. Такую программу будет сложно закрыть. Пользователь поймет суть проблемы, но избавиться от нее сможет только через снятие задачи, потому что окно приложения не будет реагировать на любые действия пользователя.

Этот побочный эффект может оказаться полезным для невидимых окон. А если код отображения главного окна программы просто удален, тогда обработчик сообщений окна останется "без работы", его тоже можно будет убрать.

Теперь добавим еще один простой, но очень интересный эффект. Будем перебирать все окна и свертывать их. Тогда функция `EnumWindowsWnd` (вызывается, когда найдено очередное окно) будет выглядеть следующим образом:

```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,      // handle to parent window
    LPARAM lParam  // application-defined value
   )
{
       ShowWindow(hwnd, SW_MINIMIZE);
       return TRUE;
}
```

Здесь в вызываемой функции `ShowWindow` в качестве второго параметра указывается флаг `SW_MINIMIZE`, который и заставляет найденное окно свернуться. Будьте осторожны при запуске программы. Функция `FindWindow` ищет все окна, в том числе и невидимые.

Примечание

Исходный код этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter3\RandMinimize.

3.2. Дрожь в ногах

Теперь усложним написанный в предыдущем разделе пример и напишем программу, которая будет перебирать все окна и изменять их размеры и положение, чтобы создавалось впечатление дрожания.

Создайте в Visual C++ новый проект типа Win32 Project. Добавьте пункт меню для вызова команды дрожания, потом найдите в исходном коде функцию `WndProc`, где обрабатываются все события окна. Между дрожаниями понадобится задержка, чтобы пользователь успел увидеть изменения, но они не казались слишком частыми, поэтому в начале функции объявляется переменная типа `HANDLE`, которая инициализируется функцией `CreateEvent`:

```
HANDLE h;
h=CreateEvent(0, true, false, "et");
```

Теперь обработчик события будет выглядеть следующим образом:

```
case ID_MYCOMMAND_VIBRATION:
       while (TRUE)
       {
              EnumWindows(&EnumWindowsWnd, 0);
              WaitForSingleObject(h,10);// Задержка в 10 миллисекунд
       }
```

Так же, как и в предыдущем примере, сначала запускается бесконечный цикл, внутри которого вызывается функция перебора всех окон и создается задержка уже знакомой нам функцией `WaitForSingleObject`. Ну что поделаешь, если я люблю использовать ее.

Самое интересное скрывается в функции `EnumWindowsWnd`, код которой вы можете увидеть в листинге 3.1.

Листинг 3.1. Код функции `EnumWindowsWnd`

```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,     // handle to parent window
    LPARAM lParam  // application-defined value
   )
{
       if (IsWindowVisible(hwnd)==FALSE)
              return TRUE;

       RECT rect;
       GetWindowRect(hwnd, &rect);
```

```
        int index=rand()%2;
        if (index==0)
        {
               rect.top=rect.top+3;
               rect.left=rect.left+3;
        }
        else
        {
               rect.top=rect.top-3;
               rect.left=rect.left-3;
        }

        MoveWindow(hwnd, rect.left, rect.top, rect.right-rect.left,
        rect.bottom-rect.top, TRUE);

        return TRUE;
}
```

Теперь посмотрим на функцию-ловушку `EnumWindowsWnd`, которая будет вызываться каждый раз, когда найдено окно. Тут первой запускается функция `IsWindowVisible`, которая проверяет, является ли найденное окно видимым. Если нет, то возвращается значение `TRUE`, происходит выход из ловушки, и поиск следующего окна будет продолжен, иначе он остановится, и следующее окно не будет найдено. Если окно невидимое, то нет смысла его двигать или изменять размер.

После этого вызывается функция `GetWindowRect`. Этой функции передается в первом параметре идентификатор найденного окна, а она возвращает во втором параметре размеры этого окна в структуре `RECT`, описывающей прямоугольную область на экране с параметрами `left`, `top`, `bottom`, `right`.

После получения величины окна генерируется случайное число от 0 до 1 с помощью функции `rand`. После этого необходимо его проверить: если сгенерированное число равно 0, то увеличиваем свойства `top` и `left` структуры `rect` на 3 пиксела, иначе эти значения уменьшаем.

Изменив значения переменных структуры, в которой хранились размеры найденного окна, перемещаем это окно с помощью функции `MoveWindow`. Эта функция имеет следующие параметры:

- идентификатор окна, позицию которого надо изменить (`h`);
- новая позиция левого края (`rect.left`);
- новая позиция верхнего края (`rect.top`);
- новая ширина (`rect.Right-rect.left`);
- новая высота (`rect.Bottom-rect.top`).

Ну, и напоследок результату работы функции присваиваем значение `TRUE`, чтобы поиск продолжился.

Получается, что если запустить программу, то вы увидите дрожание всех запущенных окон. Программа будет перебирать все окна и случайным образом изменять их положение. Попробуйте посмотреть этот эффект в действии, он потрясающий, т. е. сотрясающий!



Исходный код этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter3\Vibration.

3.3. Переключение экранов

Помнится, когда появилась первая версия программы Dashboard (она была еще под Windows 3.1), меня очень сильно заинтересовала возможность переключения экранов, и я долго искал готовую WinAPI-функцию, которой достаточно указать, какой экран надо показать, и все готово. Но это оказалось не так.

Немного позже я узнал про такую же возможность в ОС Linux, где есть виртуальные консоли (экраны). Я некоторое время помучился, но написал собственную маленькую утилиту для переключения экранов под Windows 9*x*. Сейчас я воспользуюсь этим нехитрым приемом для написания небольшой программы-шутки.

Как работает переключение экранов? Сразу открою вам секрет: никакого переключения реально не происходит. Просто все видимые окна убираются с Рабочего стола за его пределы так, чтобы вы их не видели. После этого перед пользователем остается чистый Рабочий стол. Когда нужно, все возвращается обратно. Как видите, все гениальное — просто.

При переключении окна мгновенно перемещаются за границы видимости. Мы же будем перемещать все плавно, чтобы видеть, как все окна двигаются за левый край экрана. Тем самым будет создаваться эффект, будто окна убегают от нашего взора. Программа будет невидима, поэтому закрыть ее можно только снятием задачи. Самое интересное — наблюдать за этим процессом, потому что если вы не успеете снять задачу за 10 секунд, то окно Диспетчера задач тоже убежит, и придется начинать все заново.

Только вот перемещать окна надо не теми функциями, которые нам уже знакомы. Простые функции установки позиции тут не подойдут, потому что после изменения расположения каждого окна оно перерисовывается и отнимает много процессорного времени. Если у вас открыто 20 программ, то с по-

мощью функции SetWindowPos перемещение будет слишком медленным и заметным. А нам нужно перемещать все сразу и одним заходом, без задержек.

Для того чтобы лжепереключения происходили быстро, в Windows есть несколько специальных функций, которые перемещают все указанные окна сразу. Рассмотрим пример использования этих функций.

Создайте новый проект Win32 Project и перейдите в функцию _tWinMain. Воспользуйтесь листингом 3.2 и до цикла обработки сообщений напишите необходимый для перемещения окон код.

Листинг 3.2. Код перемещения окон

```
HANDLE h=CreateEvent(0, true, false, "et");

// Бесконечный цикл
while (TRUE)
{
  int windowCount;
  int index;
  HWND winlist[10000];
  HWND w;
  RECT WRct;

  for (int i=0; i<GetSystemMetrics(SM_CXSCREEN); i++)
  {
     // Считаем окна
     windowCount=0;
     w=GetWindow(GetDesktopWindow(),GW_CHILD);

     while (w!=0)
     {
       if (IsWindowVisible(w))
       {
          winlist[windowCount]=w;
          windowCount++;
       }
       w=GetWindow(w,GW_HWNDNEXT);//Искать следующее окно
     }
     // Начало сдвига
     HDWP MWStruct = BeginDeferWindowPos(windowCount);

     // Определяем окна, которые надо сдвигать
     for (int index=0; index<windowCount; index++)
```

```
    {
       GetWindowRect(winlist[index], &WRct);
       MWStruct=DeferWindowPos(MWStruct, winlist[index], HWND_BOTTOM,
                    WRct.left-10, WRct.top,
                    WRct.right-WRct.left,
                    WRct.bottom-WRct.top,
                    SWP_NOACTIVATE || SWP_NOZORDER);
    }

    // Конец сдвига
    EndDeferWindowPos(MWStruct);// Конец сдвига
  }

  WaitForSingleObject(h,2000);// Задержка в 2000 мс
}
```

В самом начале запускается бесконечный цикл с помощью вызова `while (true)`. Внутри цикла код делится на три маленькие части: сбор указателей на окна, сдвиг окон и задержка в 10 секунд. С задержкой мы уже сталкивались не один раз, и она вам уже должна быть знакома.

Сбор активных окон происходит следующим образом:

```
// Считаем окна
w=GetWindow(GetDesktopWindow(),GW_CHILD);
while (w!=0)
{
   if (IsWindowVisible(w))
   {
      winlist[windowCount]=w;
      windowCount++;
   }

   w=GetWindow(w,GW_HWNDNEXT);// Искать следующее окно
}
```

В первой строке получаем указатель первого окна на Рабочем столе и записываем его в переменную `w`. Потом начинается цикл, который будет выполняться, пока полученный указатель не станет равным нулю, т. е. пока не переберем все окна.

В этом цикле, прежде чем запомнить указатель, происходит проверка видимости окна с помощью функции `IsWindowVisible` с параметром `w`. Если окно невидимо или свернуто (функция возвращает `FALSE`), то нет смысла его перемещать, в противном случае указатель сохраняется в массиве `winlist` и увеличивается счетчик `windowCount`.

Итак, для поиска видимых окон используется функция `GetWindow`, которая может искать все окна, включая главные и подчиненные. Идентификатор найденного окна сохраняется в переменной `w`.

В данном случае для хранения указателей на окна используется массив заранее определенной длины (`HWND winlist[10000]`). В качестве длины я взял 10 000 элементов, и этого достаточно для хранения всех запущенных программ, потому что даже больше 100 окон запускать никто не будет. Да, такой вариант не очень хорош, но прошу не придираться.

В идеальном случае надо использовать динамические массивы (массив с динамически изменяемым количеством элементов), но я не стал применять их, чтобы не усложнять пример. Наша задача — посмотреть интересные алгоритмы, а вы потом можете сами доработать их до универсального вида.

После выполнения этого кода в массиве `winlist` будут храниться указатели всех запущенных и видимых программ, а в переменной `windowCount` — количество указателей в массиве.

А теперь о самом сдвиге. Он начинается с вызова API-функции `BeginDeferWindowPos`. Эта функция выделяет память для нового окна рабочего стола, куда мы будем сдвигать все видимые окна. В качестве параметра нужно указать, сколько окон мы будем двигать.

Для сдвига окна в подготовленную область памяти используется функция `DeferWindowPos`. В данный момент не происходит никаких реальных перемещений, а изменяется только информация о позиции и размерах. У этой функции следующие параметры:

- результат выполнения функции `BeginDeferWindowPos`;
- указатель на окно, которое надо переместить, — очередной элемент из массива;
- номер по порядку (после какого окна должно быть помещено указанное);
- следующие четыре параметра указывают координаты левой верхней позиции, ширину и высоту окна — получены с помощью функции `GetWindowRect`, и левая позиция для последующего сдвига уменьшена на 10;
- флаги — указываем, что не надо активировать окно и упорядочивать.

После перемещения всех окон вызывается API-функция `EndDeferWindowPos`. Вот тут окна реально перескакивают в новое место. Это происходит практически моментально. Если бы вы использовали простую API-функцию `SetWindowPos` для установки позиции каждого окна в отдельности, то обрисовка происходила бы намного медленнее.

Хороший стиль программирования подразумевает, что все переменные, требующие значительной памяти (например, объекты), должны инициализироваться и уничтожаться. Во время инициализации память выделяется, а во время уничтожения — освобождается. Если не освобождать запрошенные ресурсы, то через какое-то время компьютер начнет очень медленно работать или может потребовать перезагрузку.

В примере я создавал объект, но нигде его не уничтожал, потому что программа выполняется бесконечно, и ее работа может прерваться только по двум причинам:

1. Выключили компьютер. В этом случае, даже если мы будем освобождать память, то она никому не понадобится.
2. Сняли процесс. Если кто-то додумается до этого, то программа закончит выполнение аварийно, а память все равно не освободится, даже если вставлен соответствующий код.

Получается, что освобождать объект бесполезно. Но все же я не советую вам пренебрегать такими вещами и в любом случае выполнять уничтожение объектов. Лишняя строчка кода никому не помешает, даже если вы думаете, что она никогда не выполнится. Зато это приучит вас всегда писать качественный код.

Попробуйте запустить программу, и все окна моментально улетят влево. Попытайтесь вызвать меню (щелкнуть правой кнопкой на Рабочем столе), и оно тоже улетит влево максимум через 2 секунды. Перемещаться будут любые запущенные программы.

Мне самому так понравился пример, что я целых полчаса играл с окнами. Они так интересно исчезают, что я не мог оторваться от этого глупого занятия. Но больше всего мне понравилось тренировать себя в скорости снятия приложения. Для этого я установил задержку в 5 секунд, потом 4 и тренировал свои пальцы в быстром нажатии клавиатурной комбинации <Ctrl>+ +<Alt>+<Del> и поиске приложения, которое надо снять. Сложность в том, что окно с процессами тоже улетает, и если не успеть снять задачу, то придется повторять попытку снова.

Таким вот способом реализовано большинство программ, переключающих экраны. Во всяком случае, других методов мне не известно и готовых функций я тоже не нашел.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter3\DesktopSwitch.

3.4. Нестандартные окна

Еще в 1995 г. почти все окна были прямоугольными, и всех это устраивало. Но несколько лет назад начался самый настоящий бум создания окон неправильной формы. Любой хороший программист считает своим долгом, чтобы его программа явно выделялась среди всех конкурентов. А в Windows Vista прямоугольные компоненты стали еще более непопулярными, и почти везде есть как минимум закругление.

Лично я против нестандартных окон и использую их очень редко, потому что этот вопрос очень важен для любого коммерческого продукта. Нестандартная форма очень плохо сказывается на восприятии окна, особенно в бизнес-приложениях. На эту тему я написал статью "Неправильное оформление", которую вы можете найти в определенном файле на компакт-диске в каталоге Doc\Other.

Но иногда сталкиваешься с ситуацией, когда необходимо сделать окно красивым и произвольной формы. Да и специфика рассматриваемых в книге примеров позволяет тренировать воображение. Поэтому мы просто обязаны рассмотреть данную тему подробнее.

Для начала попробуем сделать окно овальной формы. Создайте новый проект Win32 Project и подкорректируйте функцию `InitInstance` в соответствии с листингом 3.3. Код, который надо добавить, выделен комментариями.

Листинг 3.3. Создание окна овальной формы

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   // Начало кода, который надо добавить
   HRGN FormRgn;
   RECT WRct;
```

```
    GetWindowRect(hWnd, &WRct);
    FormRgn=CreateEllipticRgn(0, 0, WRct.right-WRct.left,
              WRct.bottom-WRct.top);
    SetWindowRgn(hWnd, FormRgn, TRUE);
    // Конец добавляемого кода


    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}
```

Первым делом надо объявить две переменные:

- FormRgn типа HRGN — используется для хранения регионов, которые описывают внешний вид окна;
- WRect типа RECT — для хранения размеров и положения окна, чтобы знать область, по которой строить овал.

На следующем этапе получаем размеры окна с помощью уже знакомой функции GetWindowRect. Теперь все готово для построения овальной области. Для этого используются две функции: CreateEllipticRgn и SetWindowRgn. Рассмотрим их подробнее:

```
HRGN CreateEllipticRgn(
    int nLeftRect,      // x-координата левого верхнего угла
    int nTopRect,       // y-координата левого верхнего угла
    int nRightRect,     // x-координата правого нижнего угла
    int nBottomRect     // y-координата правого нижнего угла
   );
```

Данная функция создает регион окна (область) в виде эллипса. В качестве параметров передаются размеры эллипса.

```
int SetWindowRgn(
    HWND hWnd,        // Указатель на окно
    HRGN hRgn,        // Предварительно созданный регион
    BOOL bRedraw      // Флаг перерисовки окна
   );
```

Эта функция назначает указанному в качестве первого параметра окну созданный регион, который передается во втором параметре. Если последний параметр (флаг) установлен в TRUE, то окно после назначения нового региона будет перерисовано, иначе это придется сделать в явном виде самостоятельно. В предложенном коде после установки региона есть вызов функции

`UpdateWindow`, которая перерисовывает окно, поэтому последний параметр можно было бы установить и в `FALSE`.

Запустите приложение, и вы увидите окно овальной формы (рис. 3.3).

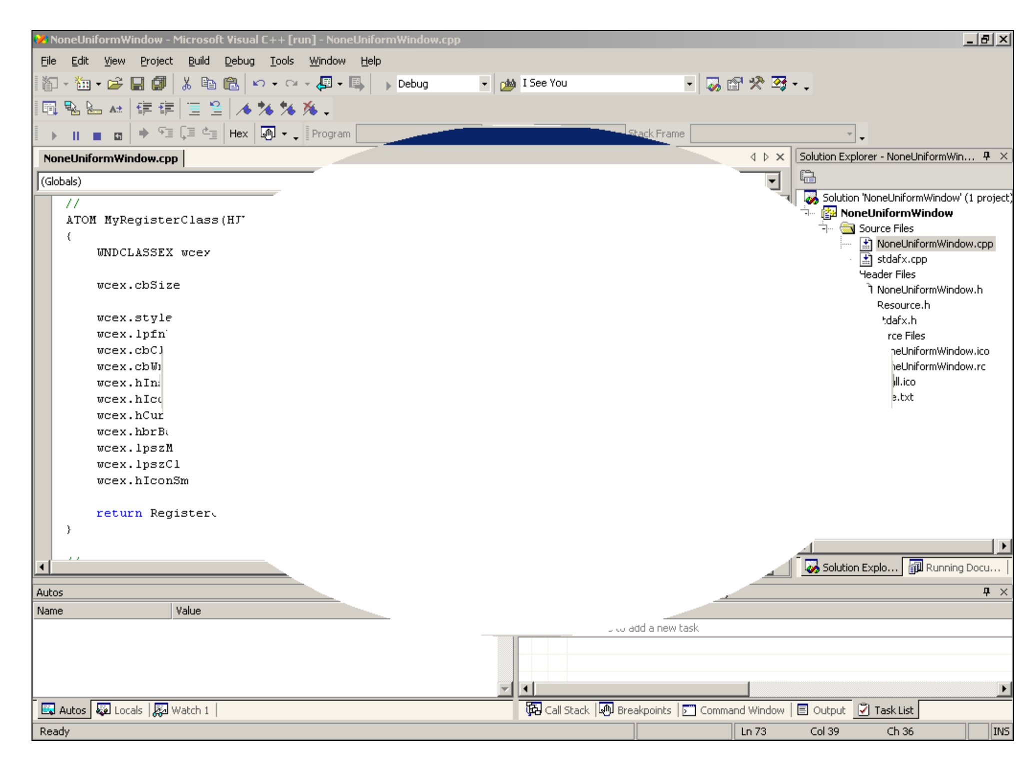


Рис. 3.3. Окно овальной формы

Теперь немного усложним задачу и попробуем создать овальное окно с прямоугольным отверстием в центре. Для этого нужно изменить код следующим образом:

```
HRGN FormRgn, RectRgn;
RECT WRct;
GetWindowRect(hWnd, &WRct);
FormRgn=CreateEllipticRgn(0,0,WRct.right-WRct.left,WRct.bottom-WRct.top);

RectRgn=CreateRectRgn(100, 100, WRct.right-WRct.left-100,
WRct.bottom-WRct.top-100);
CombineRgn(FormRgn,FormRgn,RectRgn,RGN_DIFF);
SetWindowRgn(hWnd, FormRgn, TRUE);
```

Здесь объявлены две переменные типа `HRGN`. В первой (`FormRng`) создается овальный регион функцией `CreateEllipticRgn`, а во второй — прямоугольный

с помощью функции CreateRectRgn. Для функции CreateRectRgn так же, как и при создании овального региона, указываются четыре координаты, задающие размер прямоугольника. Результат сохраняется в переменной RectRng.

После создания двух областей они объединяются с помощью функции CombineRng:

```
int CombineRgn(
    HRGN hrgnDest,              // Указатель на результирующий регион
    HRGN hrgnSrc1,              // Указатель на первый регион
    HRGN hrgnSrc2,              // Указатель на второй регион
    int fnCombineMode           // Метод комбинирования
   );
```

Эта функция комбинирует два региона, hrgnSrc1 и hrgnSrc2, и помещает результат в переменную HrgnDest.

В функции необходимо задать режим слияния (переменная fnCombineMode). Можно указать один из следующих вариантов:

- RGN_AND — объединить два региона (область перекрывания);
- RGN_COPY — копировать (копия первой области);
- RGN_DIFF — объединить разницей (удаление второй области из первой);
- RGN_OR — объединить области;
- RGN_XOR — объединить области, исключая все пересечения.

Результат работы программы вы можете увидеть на рис. 3.4.

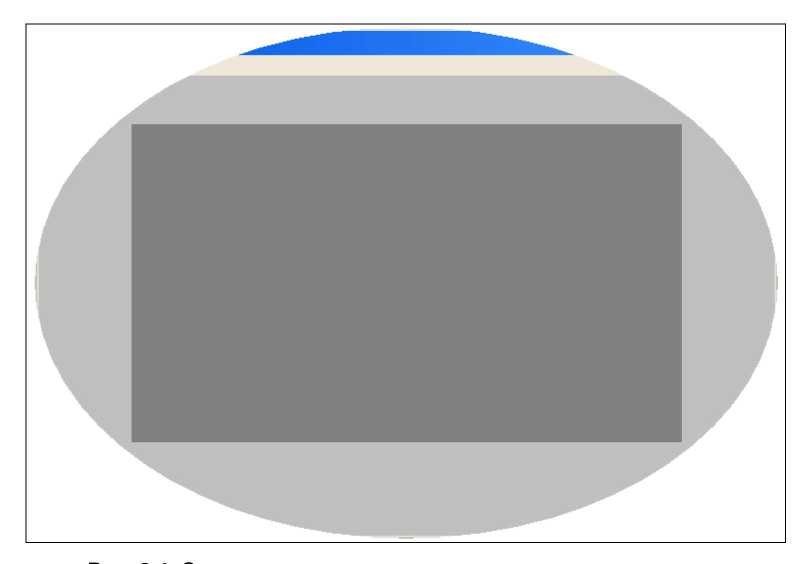


Рис. 3.4. Овальное окно с прямоугольным отверстием в центре



Исходный код этого примера вы можете найти в каталоге \Demo\Chapter3\NoneUniformWindow.

Изменять форму можно не только у окон, но и у некоторых элементов управления. Давайте рассмотрим, как это делается.

Создайте новый проект типа MFC Application. Нам сейчас не понадобится минимальный код, поэтому для упрощения воспользуемся объектной библиотекой MFC.

В Мастере создания проекта откройте раздел **Application Type** и выберите тип приложения **Dialog based** (рис. 3.5). Остальные настройки можно не менять. Я дал проекту имя None.

Рис. 3.5. Выбор типа приложения в окне Мастера создания проекта

Откройте файл ресурсов и в разделе **DIALOG** дважды щелкните по пункту **IDD_NONE_DIALOG**. Поместите в форму (рис. 3.6) один компонент `List Control`.

Теперь, чтобы с этим элементом управления можно было работать, щелкните по нему правой кнопкой мыши и выберите в появившемся меню пункт **Add Variable...**. В открывшемся окне достаточно ввести имя переменной в поле

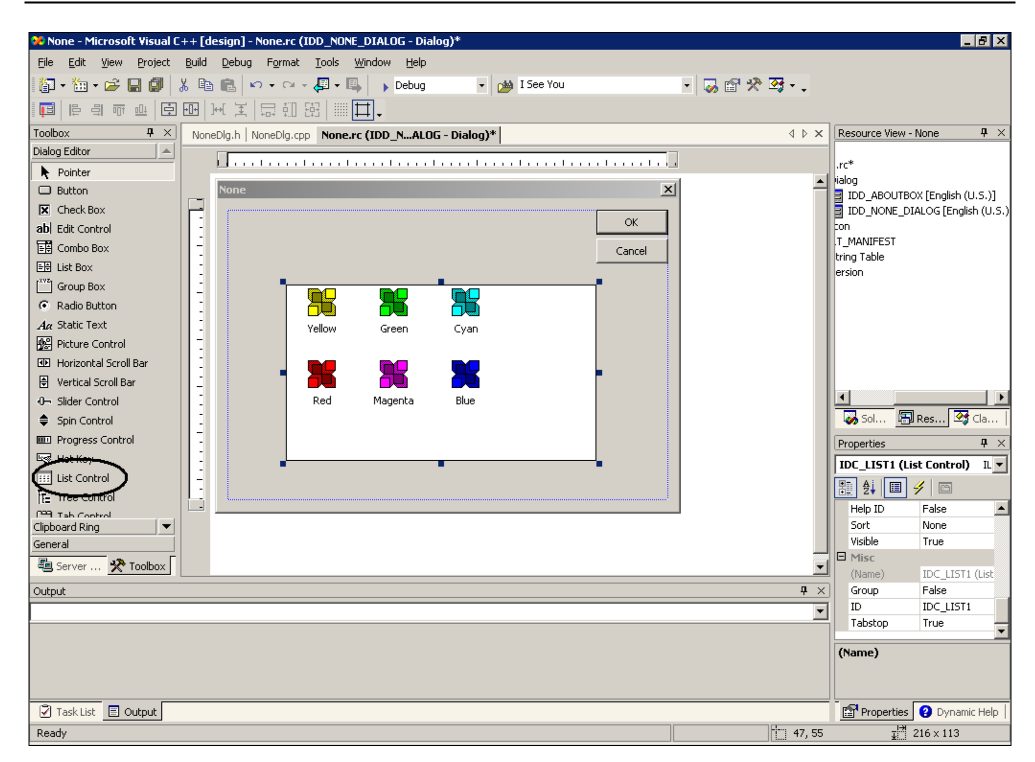


Рис. 3.6. Форма будущей программы None

Variable name. Укажите имя `ItemsList` (рис. 3.7). Нажмите кнопку **Finish**, чтобы завершить создание переменной.

Откройте файл NoneDlg.cpp и найдите здесь функцию `CNoneDlg::OnInitDialog()`. В самый конец функции, где комментарий `// TODO: Add extra initialization here`, добавьте следующий код:

```
// TODO: Add extra initialization here
RECT WRct;
HRGN FormRgn;
::GetWindowRect(ItemsList, &WRct);
FormRgn=CreateEllipticRgn(0,0,WRct.right-WRct.left,WRct.bottom-WRct.top);
::SetWindowRgn(ItemsList, FormRgn, TRUE);
```

Здесь выполняется уже знакомый код, только вместо указателя на окно используется переменная элемента управления `List Control`. Перед функциями `GetWindowRect` и `SetWindowRect` стоит знак `::`, который указывает на необходимость использования этих функций из набора WinAPI, а не MFC.

На рис. 3.8 вы можете увидеть результат работы программы.

Add Member Variable Wizard - None

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access:

public

Control variable

Variable type:

CListCtrl

Control ID:

IDC_LIST3

Category:

Control

Variable name:

ItemsList

Control type:

SysListView32

Max chars:

Min value:

Max value:

.h file:

.cpp file:

Comment (// notation not required):

Finish Cancel Help

Рис. 3.7. Окно создания переменной для элементов управления

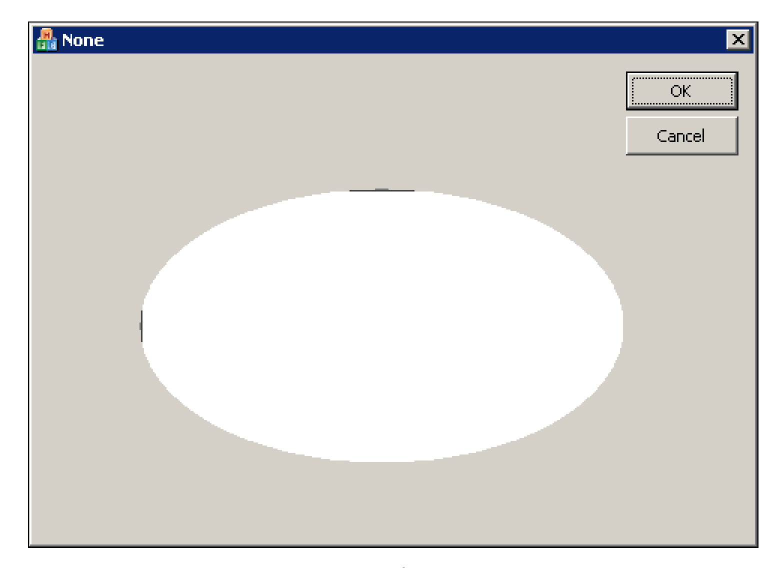


Рис. 3.8. Результат работы программы None



Исходный код этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter3\None.

3.5. Безбашенные окна

Вы уже научились делать окна на основе простых фигур (овал, прямоугольник) и их сочетания, и теперь нам предстоит узнать, как создать окно совершенно произвольной формы. Такое окно уже невозможно будет сделать парой комбинаций. Тут уже нужны более сложные манипуляции. Но в результате вы увидите, что ничего сверх школьной программы математики в наших расчетах не будет.

На рис. 3.9 представлена картинка с красным фоном. Я понимаю, что я очень веселый, и вы просто не сможете увидеть красный фон на черно-белом рисунке, но все же представьте все вокруг Шрека красным. Как сделать окно, которое будет содержать это изображение, а красный цвет сделать прозрачным, чтобы окно приняло форму картинки? Не будем это делать с помощью комбинирования регионов из простых фигур (хотя такое и возможно, просто их понадобится много). Нет, мы не будем распознавать фигуры и писать сложные алгоритмы, задача решается очень просто — использованием только прямоугольников.



Рис. 3.9. Маска для будущего окна

В WinAPI есть регионы типа полигонов, но это не сильно упростит задачу, ибо распознавать образы — задача не из легких.

Итак, создадим один большой регион, который опишет наше изображение. Код будет универсальным, и вы сможете подставить вместо моей картинки что-то другое. Как мы будем это делать? Очень просто.

Что представляет собой изображение? Это двумерный массив точек. Нам нужно взглянуть на каждую строку — как на отдельный элемент региона, т. е. мы будем строить прямоугольный регион по каждой строке, а потом скомбинируем все вместе. Алгоритм будет выглядеть следующим образом:

1. Сканируем строку и находим первый пиксел, отличный от прозрачного. Это будет начало нашего прямоугольника (координата `X1`).
2. Сканируем остаток строки, чтобы найти границу прозрачности (последний непрозрачный пиксел, координата `X2`). Если прозрачных пикселов нет, то регион строится до конца строки.
3. Координату `Y1` принимаем равной номеру строки, а `Y2` — равной `Y1+1`. Таким образом, высота прямоугольника, описывающего одну строку, равна одному пикселу.
4. Строим регион по найденным координатам.
5. Переходим к следующей строке.
6. Объединяем созданные регионы и назначаем их окну.

Это упрощенный алгоритм, а в реальной жизни может быть два и более прямоугольных региона на одну строку, когда в середине строки встречаются прозрачные области.

Описанный алгоритм реализован в виде кода на языке C++ и представлен в листинге 3.4. Чуть позже я его рассмотрю.

А пока поговорим о том, каким должен быть графический файл. Это может быть любой Windows BITMAP-файл. Его размеры можно рассчитать в зависимости от величины изображения, но в данном случае ограничимся заранее определенными значениями (200×200 пикселов). Самостоятельно попробуйте сделать код еще более универсальным.

В самой картинке цвет пиксела с координатами 0:0 считается прозрачным, поэтому при подготовке изображения надо учесть, что все прозрачные области в окне должны быть окрашены этим цветом. Это лучше, чем использовать заранее определенный цвет, потому что он может быть необходим изображению. А вот левый верхний угол чаще всего свободен, но даже если и нет, один пиксел всегда можно сделать прозрачным (т. е. не учитывать). На общую картину он не повлияет.

Создайте новый проект Win32 Project. Найдите функцию `InitInstance` и измените функцию создания окна следующим образом:

```
hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, 200, 200, NULL, NULL, hInstance, NULL);
```

Здесь числа 200 указывают ширину и высоту окна. Если ваше изображение другого размера, то измените эти значения.

Кроме того, нужно убрать меню, потому что использовать его нет смысла. Для этого найдите функцию `MyRegisterClass` и строку, где изменяется свойство `wcex.lpszMenuName`. Ему нужно присвоить нулевое значение:

```
wcex.lpszMenuName = 0;
```

В разделе глобальных переменных нужно добавить следующие две переменные:

```
HBITMAP maskBitmap;
HWND hWnd;
```

Первая переменная будет использоваться для хранения изображения, а вторую мы уже неоднократно использовали для хранения указателя на окно. Объявление переменной `hWnd` надо удалить из функции `InitInstance`, чтобы использовать глобальную переменную.

Теперь измените функцию `_tWinMain` в соответствии с листингом 3.4, и можно считать, что ваша программа готова.

Прежде чем запускать программу, просто откомпилируйте ее и перейдите в папку, в которой находятся исходные коды. Если вы будете работать с программой в режиме отладки, то у вас должна появиться в этом месте подпапка Debug, иначе — Release. Поместите в нее подготовленный графический файл. Если файла не будет, то произойдет ошибка, потому что после запуска программа будет искать файл в текущей папке, а текущей будет Debug или Release. Если вы компилировали в разных режимах, то лучше поместить файл сразу в обе папки, на случай повторного переключения режима.

Листинг 3.4. Создание окна произвольной формы на основе маски

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
        // TODO: Place code here.
       MSG msg;
       HACCEL hAccelTable;

       // Initialize global strings
       LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
       LoadString(hInstance, IDC_MASKWINDOW, szWindowClass,
              MAX_LOADSTRING);
       MyRegisterClass(hInstance);
```

```
// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
       return FALSE;
}

hAccelTable = LoadAccelerators(hInstance,
       (LPCTSTR)IDC_MASKWINDOW);

// Следующий код вы должны добавить
// Сначала убираем обрамление
int Style;
Style = GetWindowLong(hWnd, GWL_STYLE);
Style=Style || WS_CAPTION;
Style=Style || WS_SYSMENU;
SetWindowLong(hWnd, GWL_STYLE, Style);
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

// Загружаем картинку
maskBitmap = (HBITMAP)LoadImage( NULL, "mask.bmp",
                        IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE );
if ( !maskBitmap ) return NULL;

// Описание необходимых переменных
BITMAP bi;
BYTE bpp;
DWORD TransPixel;
DWORD pixel;
int startx;
INT i, j;

// Создаем пустой регион
HRGN Rgn, ResRgn = CreateRectRgn(0, 0, 0, 0);
// Получаем информацию об объекте
GetObject(maskBitmap, sizeof(BITMAP), &bi);
// Определяем размер необходимого буфера и выделяем его
bpp = bi.bmBitsPixel >> 3;
BYTE *pBits = new BYTE[bi.bmWidth * bi.bmHeight * bpp];

// Получаем битовый массив
int  p = GetBitmapBits(maskBitmap,
       bi.bmWidth * bi.bmHeight * bpp, pBits);
```

```
// Определяем цвет прозрачного символа
TransPixel = *(DWORD*)pBits;

TransPixel <<= 32 — bi.bmBitsPixel;

// Цикл, перебирающий все строки
for (i = 0; i < bi.bmHeight; i++)
{
   startx=-1;
   // Цикл, перебирающий все пикселы в строке
   for (j = 0; j < bi.bmWidth; j++)
   {
      // В pixel копируем цвет текущего пиксела
      pixel = *(DWORD*)(pBits + (i * bi.bmWidth +
                    j) * bpp) << (32 — bi.bmBitsPixel);
      // является ли пиксел прозрачным
      if (pixel != TransPixel)
      {
         // Пиксел не прозрачный
         // Есть ли уже запомненное место?
         if (startx<0)
         {
             // Нет, просто запоминаем начало
             // будущего прямоугольника
             startx = j;
         } else
         // Достигли ли мы конца строки
         if (j == (bi.bmWidth — 1))
         {
             // Создать прямоугольный регион и скомбинировать
             // с базовым
             Rgn=CreateRectRgn(startx, i, j, i+1);
             CombineRgn(ResRgn, ResRgn, Rgn, RGN_OR);
             startx=-1;
         }
      } else if (startx>=0)
      {
            // Создаем область до текущей позиции
             Rgn = CreateRectRgn(startx, i, j, i + 1);
             CombineRgn(ResRgn, ResRgn, Rgn, RGN_OR);
             startx=-1;
      }
   }
}
delete[] pBits;
```

```
        // Назначить новый регион
        SetWindowRgn(hWnd, ResRgn, TRUE);
        InvalidateRect(hWnd, 0, false);
        // Конец добавляемого кода

        // Main message loop:
        ...
        ...
}
```

В самом начале из окна убирается системное меню и обрамление. После этого загружается картинка, уже знакомой функцией `LoadImage`. Изображение читается из файла, поэтому первый параметр `NULL`, второй содержит имя файла, а в последнем указан флаг `LR_LOADFROMFILE`. Так как мы указали только имя файла (без полного пути), поэтому программа будет искать его в том же каталоге, где находится программа. Именно поэтому мы должны были скопировать mask.bmp в папку Debug или Release.

Необходимо проверить наличие файла изображения. Если переменная `maskBitmap` равна нулю, то картинка не была найдена, и произойдет выход из программы:

```
if (!maskBitmap) return NULL;
```

Это обязательная проверка, потому что дальнейшее обращение к памяти, где должны быть данные, приведет к ненужной ошибке.

После этого начинается самое интересное — анализ картинки и поиск прямоугольных областей для создания формы. Для начала создаем пустой прямоугольный регион с помощью `CreateRectRgn` и указанием нулевых позиций и размеров. Это пустой регион, который ни на что не влияет, зато к нему можно будет добавлять то, что мы отсканируем.

Следующей строкой с помощью функции `GetObject` получаем информацию о графическом объекте. У этой функции три параметра:

- загруженная картинка;
- размер структуры, указанной в третьем параметре;
- структура, в которую будет записан результат работы.

Для чего нам нужна эта информация? Из нее возьмем размеры картинки. Да, мы могли прописать размеры жестко и избежать применения этой функции, но тогда мы теряем в универсальности. Помимо размеров картинки мы определяем количество битов на пиксел, которые описывают картинку. Вся эта информация помогает нам определить размер необходимого массива, в который будут загружаться данные.

Следующая функция `GetBitmapBits` позволяет получить битовый массив картинки. И эта функция получает три параметра. Ну что поделаешь, если разработчик любит магическое число 3:

- картинка, битовые данные которой нужны;
- размер данных в буфере под данные;
- подготовленный буфер, куда функция скопирует битовые данные картинки.

Битовый массив получили, и все готово для начала сканирования, поэтому запускаем два цикла. Первый (основной), в котором будем перебирать строки, а второй (вложенный) — для перебора пикселов в текущей строке.

Внутри цикла получаем текущий пиксел и проверяем его цвет. Если он не прозрачный, то нужно проверить, была ли запомнена какая-то начальная позиция (началась ли уже прямоугольная область). Если нет, то запоминаем текущую точку — как начало. Иначе нужно проверить, достигли ли мы конца строки. Если да, то создаем прямоугольную область до конца текущей строки.

Если текущий пиксел является прозрачным, то нужно проверить, есть ли что-то запомненное? Если да, то это первый прозрачный пиксел, после закрашенной области, и нужно создать прямоугольную область.

Таким образом, сканируя строку за строкой, мы постепенно формируем форму будущего окна.



Рис. 3.10. Окно в форме рисунка

Если вы сейчас запустите пример, то увидите окно, как на рис. 3.10. Окно действительно приняло форму изображения, но оно пустое. Был создан только регион, но в самом окне ничего не нарисовано. Чтобы содержимое окна наполнить изображением картинки, надо в обработчик события `WM_PAINT`

функции `WndProc` добавить следующий код (полный код примера см. на компакт-диске):

```
case WM_PAINT:
       hdc = BeginPaint(hWnd, &ps);
       // TODO: Add any drawing code here...
       hdcBits=::CreateCompatibleDC(hdc);
       SelectObject(hdcBits, maskBitmap);
       BitBlt(hdc, 0, 0, 200, 200, hdcBits, 0, 0, SRCCOPY);
       DeleteDC(hdcBits);
       EndPaint(hWnd, &ps);
       break;
```

Здесь просто выводится изображение точно так же, как при рисовании кнопки **Пуск**. Вот теперь программа закончена, и вы можете увидеть результат ее работы на рис. 3.11.

Рис. 3.11. Приложение с окном произвольной формы



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter3\MaskWindow.

3.6. Перемещение окна за любую область

С помощью программы, описанной в *разд. 3.5*, можно получить окно произвольной формы, но оно будет иметь один недостаток — его нельзя передвигать. Это потому, что у него нет обрамления и системного меню, с помощью которых и происходит перемещение окна по Рабочему столу. У этого окна есть только рабочая область, и это усложняет задачу.

Чтобы избавиться от этого недостатка, мы должны научить нашу программу двигать окно по щелчку мыши в любой его зоне. Есть два способа решения этой проблемы:

- при нажатии кнопки в рабочей области можно обмануть систему и заставить ее поверить, что щелчок был произведен по заголовку окна. Тогда ОС сама будет двигать окно. Такое решение самое простое, и проблема решается одной строкой кода, но в реальной работе это неудобно, поэтому я даже не буду его рассматривать;
- самостоятельно перемещать окно. Программного кода будет намного больше, но зато он будет универсальным и гибким.

Для этого надо поймать и обработать три события окна:

1. Пользователь нажал кнопку мыши. Тогда необходимо сохранить текущую позицию курсора и запомнить в какой-нибудь переменной это событие. В нашем примере это будет глобальная переменная `dragging` типа `bool`. Помимо этого нужно захватить мышь, чтобы все ее события посылались нашему окну, пока мы перемещаем его. Для этого служит функция `SetCapture`, которой надо передать в качестве параметра указатель на окно. Это необходимо для того, чтобы в случае выхода указателя за пределы рабочей области программа все равно получала сообщения о передвижении мыши.
2. Перемещение мыши. Если переменная `dragging` установлена в `true`, то пользователь нажал кнопку мыши и двигает окно. В этом случае надо подкорректировать положение окна в соответствии с новым положением курсора. Иначе это просто движение мыши поверх окна.
3. Пользователь отпустил кнопку мыши. В этот момент необходимо присвоить переменной `dragging` значение `false` и освободить курсор с помощью функции `ReleaseCapture`.

Воспользуйтесь примером из предыдущего раздела. Откройте его и найдите функцию `WndProc`, сгенерированную мастером при создании проекта. Добавьте в нее код из листинга 3.5, выделенный комментариями.

В раздел глобальных переменных добавьте следующие две переменные:

```
bool dragging=false;
POINT MousePnt;
```

Листинг 3.5. Код перетаскивания мыши

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;
   HDC hdcBits;

   RECT wndrect;
   POINT point;

   switch (message)
   {
   case WM_COMMAND:
      ...
   case WM_PAINT:
      ...
   case WM_DESTROY:
      PostQuitMessage(0);
      break;

   ////////////////////////////////////////////
   // Начало кода, который надо добавить
   ////////////////////////////////////////////

   // Следующий код обрабатывает событие,
   // когда нажата левая кнопка мыши
   case WM_LBUTTONDOWN:
      GetCursorPos(&MousePnt);
      dragging = true;
      SetCapture(hWnd);
      break;
   // Следующий код обрабатывает событие,
   // когда курсор мыши движется по экрану
   case WM_MOUSEMOVE:
      if (dragging) // Если нажата кнопка, то...
      {
```

```
          // Получить текущую позицию курсора
          GetCursorPos(&point);
          // Получить текущие размеры окна
          GetWindowRect(hWnd, &wndrect);

          // Откорректировать положение окна
          wndrect.left = wndrect.left + (point.x — MousePnt.x);
          wndrect.top  = wndrect.top  + (point.y — MousePnt.y);

          // Установить новые размеры окна
          SetWindowPos(hWnd, NULL, wndrect.left, wndrect.top,
                  0, 0, SWP_NOZORDER | SWP_NOSIZE);

          // Запоминаем текущую позицию курсора
          MousePnt=point;
      }
      break;
   // Следующий код обрабатывает событие,
   // когда левая кнопка мыши отпущена
   case WM_LBUTTONUP:
       if (dragging)
       {
          dragging=false;
          ReleaseCapture();
       }
   ////////////////////////////////////////////
   // Конец кода, который надо добавить
   ////////////////////////////////////////////
   default:
       return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Все функции в примере вам уже должны быть знакомы, но программа получилась большая, и я написал подробные комментарии, чтобы вам легче было в ней разобраться.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter3\MaskWindow2.

3.7. Подсматриваем пароли

В большинстве программ вводимый пароль отображается звездочками. Это делается для того, чтобы никто из окружающих не увидел, что вы набираете при входе в приватную область своего компьютера. А что делать, если вы ввели пароль в программу и забыли?

Как увидеть пароль, спрятанный под звездочками? Для этого есть много разных специальных программ. Но вы же не думаете, что я буду вас отправлять к ним в своей книге? Конечно же, сейчас мы разберем, как самостоятельно написать подобную программу.

Тут нужно отметить, что программа будет работать, но далеко не везде. Дело в том, что она может показать пароль только тогда, когда он реально спрятан под звездочками, но это далеко не всегда так. Некоторые программы, в которых безопасность стоит на первом месте (в том числе и сама ОС Windows), не сохраняют пароли в системе. Вместо этого система хранит только хэш-сумму, а в полях ввода отображают бутафорию в виде звездочек. Так что убирать такие звездочки бесполезно.

Наш пример будет состоять из двух частей. Первый файл (запускаемый) будет загружать другой файл (динамическую библиотеку) в память. Эта библиотека будет регистрироваться в системе в качестве обработчика системных сообщений, который будет ожидать, когда пользователь щелкнет в интересующем его окне правой кнопкой мыши. Как только такое событие произойдет, мы сразу должны будем получить текст этого окна и конвертировать его из звездочек в обычный текст. На первый взгляд, все выглядит достаточно сложным, но реально вы сможете реализовать все за десять минут.

3.7.1. Динамическая библиотека для расшифровки паролей

Для этого примера я написал DLL-файл, процесс создания которого будет сейчас расписан на ваших глазах. Создайте новый проект Win32 Project в Visual C++ и назовите его OpenPassDLL. В Мастере настроек приложения (Application Settings) выберите тип приложения — DLL (рис. 3.12).

В новом проекте у вас будет только один файл OpenPassDLL.cpp (не считая стандартного stdafx.cpp), но заголовочного файла для него не будет. В заголовочных файлах принято писать все объявления, а нам они понадобятся, поэтому давайте создадим такой файл. Для этого щелкните правой кнопкой мыши в окне **Solution Explorer** по пункту **Header Files** и в появившемся меню выберите пункт **Add/Add New Item**. Перед вами должно открыться окно, как на рис. 3.13. Выберите в правой части окна тип файла **Header File (.h)**, а в

Рис. 3.12. Окно Мастера настроек нового приложения DLL

Рис. 3.13. Окно создания заголовочного файла

поле **Name** укажите **OpenPassDLL.h**. Нажмите кнопку **Open**, чтобы добавить новый файл к проекту.

Щелкните дважды по созданному файлу, чтобы открыть его в редакторе, и напишите в нем следующий код:

```
// Macro for DLL exports in Win32, replaces Win16 __export
// (Макрос для экспорта DLL в Win32 вместо 16-разрядной версии)
#define DllExport extern "C" __declspec(dllexport)

// Prototype
// (Прототип)
DllExport void RunStopHook(bool State, HINSTANCE hInstance);
```

Сначала описывается макрос `DllExport`, с помощью которого можно указывать, какие процедуры будут экспортироваться (можно вызывать из других приложений).

Во второй строке кода описывается сама экспортируемая функция. Как видите, объявление похоже на реализацию, но отсутствует сам код, а есть только название и передаваемые значения. Сама функция должна быть написана в файле OpenPassDLL.cpp.

Переходим к файлу OpenPassDLL.cpp. Его содержимое вы можете увидеть в листинге 3.6. Повторите этот код в своем файле и внимательно изучите.

Листинг 3.6. Код файла OpenPassDLL.cpp

```
// OpenPassDLL.cpp : Defines the entry point for the DLL application
// OpenPassDLL.cpp : Определяет точку входа для DLL-приложения

#include <windows.h>
#include "stdafx.h"
#include "OpenPassDLL.h"

HHOOK SysHook;
HWND Wnd;
HINSTANCE hInst;

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                     )
{
    hInst=(HINSTANCE)hModule;
```

```
    return TRUE;
}

LRESULT CALLBACK SysMsgProc(
    int code,           // hook code (код ловушки)
    WPARAM wParam,      // flag (флаг)
    LPARAM lParam       // address of structure with message
                        // (адрес структуры с сообщением)
   )
{
    // Передать сообщение другим ловушкам в системе
    CallNextHookEx(SysHook, code, wParam, lParam);

    //Проверяю сообщение
    if (code == HC_ACTION)
    {
       // Получаю идентификатор окна, сгенерировавшего сообщение
       Wnd=((tagMSG*)lParam)->hwnd;

       // Проверяю тип сообщения
       // Если была нажата правая кнопка мыши
       if (((tagMSG*)lParam)->message == WM_RBUTTONDOWN)
       {
           SendMessage(Wnd, EM_SETPASSWORDCHAR, 0, 0);
           InvalidateRect(Wnd, 0, true);
       }
    }

    return 0;
}

///////////////////////////////////////////////////////////////////
DllExport void RunStopHook(bool State, HINSTANCE hInstance)
{
   if (true)
      SysHook = SetWindowsHookEx(WH_GETMESSAGE,
               &SysMsgProc, Inst, 0);
   else
      UnhookWindowsHookEx(SysHook);
}
```

Разберем подробно исходный код динамической библиотеки. В самом начале подключаются заголовочные файлы. OpenPassDLL.h — это созданный нами файл, в котором объявлены макрос и функция (будет экспортирована).

Далее идет определение глобальных переменных библиотеки. Их будет три:

- SysHook — идентификатор ловушки системных сообщений;
- Wnd — указатель на окно (со звездочками), в котором щелкнул пользователь. Эту переменную можно было сделать и локальной, но я решил ее вынести в глобальную область для последующего использования;
- HInst — идентификатор экземпляра библиотеки.

Описания закончены. В области программы первой идет функция DllMain. Это стандартная функция, которая выполняется при загрузке библиотеки. В ней можно производить действия по начальной инициализации. В нашем случае ничего инициализировать не надо, но в качестве первого параметра этой функции мы получаем экземпляр библиотеки, который сохраняем в переменной hInst.

Теперь рассмотрим функцию RunStopHook. Она будет запускать и останавливать системную ловушку. В качестве параметров нужно передать два значения:

- логический параметр — значение true, если надо запустить ловушку, иначе — остановить;
- идентификатор экземпляра приложения, вызвавшего эту функцию, — пока не будем использовать этот параметр.

Если в качестве первого параметра передано значение true, то регистрируется ловушка, которая будет принимать все сообщения Windows на себя. Для этого используется функция SetWindowsHookEx. У этой функции должно быть четыре параметра:

- тип ловушки — в данном случае WH_GETMESSAGE;
- указатель на функцию, которой будут пересылаться сообщения Windows;
- указатель на экземпляр приложения — переменная, в которой сохранен экземпляр библиотеки;
- идентификатор потока — если указан ноль, то используются все существующие потоки.

В качестве второго параметра указано имя функции SysMsgProc. Она также описана в этой библиотеке. Но ее мы рассмотрим чуть позже.

Значение, которое возвращает функция SetWindowsHookEx, сохраняется в переменной SysHook. Оно понадобится при отключении ловушки.

Если процедура RunStopHook получила в качестве параметра значение false, то нужно отключить ловушку. Для этого вызывается процедура UnhookWindowsHookEx, которой передается значение переменной SysHook. Это то значение, которое было получено при создании ловушки.

Теперь давайте посмотрим на функцию SysMsgProc, которая будет вызываться при наступлении системных событий.

В первой строке пойманное сообщение передается следующей ловушке, установленной в системе с помощью функции CallNextHookEx. Если этого не сделать, то другие официально зарегистрированные обработчики не смогут узнать о наступившем событии, и система будет работать некорректно.

Далее проверяется тип полученного сообщения. Нам нужно обрабатывать событие нажатия кнопки мыши, значит, параметр code должен быть равен HC_ACTION; сообщения другого типа нам нет смысла обрабатывать.

После этого определяется окно, сгенерировавшее событие, и проверяется тип события. Указатель на окно можно получить так: ((tagMSG*)lParam)->hwnd. На первый взгляд, запись абсолютно непонятная, но попробуем в ней разобраться. Основа этой записи — переменная lParam, которая получена в качестве последнего параметра нашей функции-ловушки SysMsgProc. Запись ((tagMSG*)lParam) обозначает, что структура типа tagMSG находится по адресу памяти, указатель на который передан через параметр lParam. У этой структуры есть параметр hwnd, в котором находится указатель на окно, сгенерировавшее сообщение.

Следующим этапом проверяется событие нажатия кнопки мыши: если была нажата правая кнопка мыши, то в этом окне нужно убрать звездочки. Для этого проверяется содержимое поля message все той же структуры ((tagMSG*)lParam).

Если это свойство равно WM_RBUTTONDOWN, то нажата правая кнопка мыши, и надо убрать звездочки. Для этого окну посылается сообщение SendMessage со следующими параметрами:

- Wnd — окно, которому предназначено сообщение;
- EM_SETPASSWORDCHAR — тип сообщения. Этот тип говорит о том, что надо изменить символ, который будет использоваться для того, чтобы спрятать пароль;
- 0 — новый символ, при отправке которого текущий символ-маска просто исчезнет и будет восстановлен реальный текст;
- 0 — зарезервировано.

Напоследок вызывается функция InvalidateRect, которая заново прорисует указанное в первом параметре окно. Это все то же окно, в котором произведен щелчок. Во втором параметре указывается область, которую надо прорисовать, значение 0 равносильно прорисовке всего окна. Если последний параметр true, то надо перерисовать и фон.



Исходный код библиотеки вы можете найти на компакт-диске в каталоге \Demo\Chapter3\OpenPassDLL.

3.7.2. Программа расшифровки пароля

Теперь напишем программу, которая будет загружать библиотеку и запускать ловушку. Для этого создайте новый проект Win32 Project типа **Windows Application**. В нем надо только подкорректировать функцию `_tWinMain`, как в листинге 3.7.

Листинг 3.7. Загрузка DLL-библиотеки и запуск ловушки

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
   ...

   hAccelTable = LoadAccelerators(hInstance,
              (LPCTSTR)IDC_OPENPASSTEST);

   ///////////////////////////////////////
   // Следующий код необходимо добавить
   LONG      lResult;
   HINSTANCE hModule;

   // Создаем новый указатель на функцию
   typedef void (RunStopHookProc)(bool, HINSTANCE);

   RunStopHookProc* RunStopHook = 0;

   // Load DLL file  (Чтение DLL-библиотеки)
   hModule = ::LoadLibrary("OpenPassDLL.dll");

   // Получить адрес функции в библиотеке
   RunStopHook = (RunStopHookProc*)::GetProcAddress(
            (HMODULE) hModule, "RunStopHook");

   // Выполнить функцию
   (*RunStopHook)(true, hInstance);
```

```
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
       if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
       {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
       }
    }

    (*RunStopHook)(false, hInstance);
    FreeLibrary(hModule);

    return (int) msg.wParam;
}
```

Так как функция описана в динамической библиотеке, а использоваться будет в другой программе, то необходимо указать тип вызываемой функции. Если что-то указать неправильно, то вызов будет невозможен. Описание функции делается так:

```
typedef void (RunStopHookProc)(bool, HINSTANCE);
```

Таким образом, описывается тип функции `RunStopHookProc`, которая ничего не возвращает, но принимает в качестве параметров два значения типа `bool` и `HINSTANCE`. В следующей строке объявляется переменная `RunStopHook` описанного типа, и ей присваивается значение 0.

Теперь необходимо загрузить динамическую библиотеку. Для этого есть функция `LoadLibrary`, которой нужно только передать имя файла или полный путь. Можно и относительный путь к библиотеке или только имя. В примере указано лишь имя файла, поэтому библиотека должна находиться в одном каталоге с запускаемым файлом, либо ее нужно разместить в каталоге, доступном для Windows (один из системных каталогов).

Загрузив библиотеку, надо определить адрес, по которому расположена функция `RunStopHook`, чтобы ее можно было использовать. Для этого существует функция `GetProcAddress`, которой нужно передать указатель на искомую библиотеку и название функции. Результат сохраняется в переменной `RunStopHook`.

Вот теперь все готово, и можно запускать функцию-ловушку. Это делается не совсем обычным способом:

```
(*RunStopHook)(true, hInstance);
```

Дальше запускается цикл обработки сообщений, в котором ничего изменять не надо. Но по выходе из программы следует остановить ловушку и выгру-

зить динамическую библиотеку. Мы все же хорошие программисты и должны чистить за собой. Это делается следующим образом:

```
(*RunStopHook)(false, hInstance);
FreeLibrary(hModule);
```



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter3\OpenPassTest.

Для тестирования примера поместите в одну папку исполняемый файл и динамическую библиотеку. Запустите программу и щелкните правой кнопкой мыши в поле ввода пароля. Звездочки (или любые другие символы) моментально превратятся в реальный текст.

На рис. 3.14 можно увидеть пример работы этой программы. Здесь изображено стандартное окно Windows 2000 для смены пароля. В первом поле вы видите пароль, который расшифрован нашей программой. Во втором поле, в котором нужно ввести подтверждение пароля, остались звездочки.

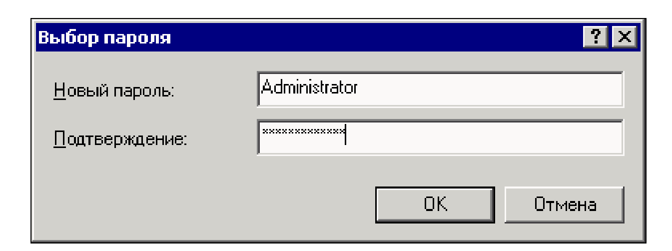


Рис. 3.14. Пример работы программы OpenPassTest

3.7.3. От пользы к шутке

Этот пример очень легко превратить в шуточный. Достаточно только поменять в динамической библиотеке пару параметров, и работа программы изменится. Попробуем обрабатывать нажатие левой кнопки мыши и не уничтожать символ пароля, а устанавливать его. В этом случае при любом щелчке пользователя весь текст будет замещаться установленным символом. В листинге 3.8 показано, как будет выглядеть функция `SysMsgProc`.

Листинг 3.8. Ловушка сообщений, в которой любой текст замещается символом "d"

```
LRESULT CALLBACK SysMsgProc(
    int code,              // hook code
    WPARAM wParam,         // removal flag
    LPARAM lParam          // address of structure with message
   )
```

```
{
   // Передать сообщение другим ловушкам в системе
   CallNextHookEx(SysHook, code, wParam, lParam);

   // Проверяю сообщение
   if (code == HC_ACTION)
   {
      // Получаю идентификатор окна, сгенерировавшего сообщение
      Wnd=((tagMSG*)lParam)->hwnd;

      // Проверяю тип сообщения
      // Если была нажата левая кнопка мыши
      if (((tagMSG*)lParam)->message == WM_LBUTTONDOWN)
      {
           SendMessage(Wnd, EM_SETPASSWORDCHAR, 100, 0);
           InvalidateRect(Wnd, 0, true);
      }
   }
   return 0;
}
```

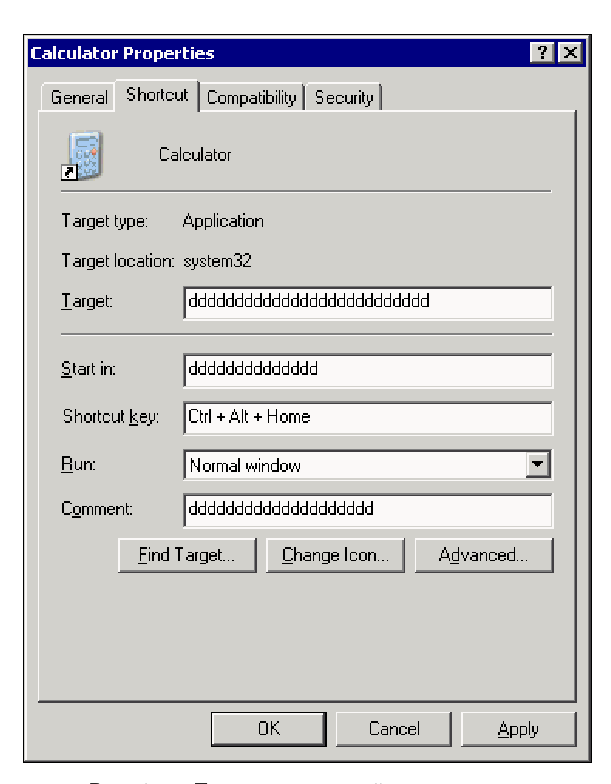


Рис. 3.15. Превращение свойств документа

Здесь проверяется нажатие левой кнопки мыши, и функция `SendMessage` отправляет в качестве третьего параметра число 100, что соответствует символу "d". Можно указать код любого другого символа. Результат: в каком поле пользователь ни щелкнет мышью, весь текст заместится указанным символом. На рис. 3.15 показано окно свойств документа программы MS Word, в котором вся информация отображается символом "d".



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter3\SetPassDLL.

3.8. Мониторинг исполняемых файлов

Часто в жизни возникают ситуации, когда необходимо определить, какие программы запускает пользователь и сколько времени он работает. Этот вопрос интересует не только хакеров, но и администраторов сетей, и руководителей предприятий.

Хакер может ожидать, когда запустится определенная программа, чтобы произвести с ней какие-нибудь манипуляции. Администратора сети интересует, что сделал пользователь, прежде чем завис компьютер. Начальника интересует использование рабочего времени.

Именно на этих задачах мне пришлось разобраться, как отследить, какие программы запускаются и как долго находятся в рабочем состоянии.

Данная проблема решается достаточно просто, и программа будет похожа на разгадывание паролей: так же необходимо создать ловушку, которая будет отслеживать определенные системные сообщения. В предыдущем примере ловушка устанавливалась с помощью API-функции `SetWindowsHookEx`, и регистрировались сообщения типа `WH_GETMESSAGE`. Если этот параметр изменить на `WH_CBT`, то такая ловушка сможет фиксировать следующие сообщения:

- `HCBT_ACTIVATE` — приложение активизировалось;
- `HCBT_CREATEWND` — создано новое окно;
- `HCBT_DESTROYWND` — уничтожено существующее окно;
- `HCBT_MINMAX` — окно свернули или развернули на весь экран;
- `HCBT_MOVESIZE` — окно переместили или изменили размер.

Таким образом, динамическая библиотека для мониторинга исполняемых программ должна соответствовать коду в листинге 3.9. Пока остановимся на поиске событий без их обработки. В реальном приложении может понадобиться сохранение полученных событий и названий окон, с которыми рабо-

тал пользователь, в каком-нибудь файле. Впоследствии по этой информации можно легко узнать, с чем и сколько работал пользователь.

Листинг 3.9. Библиотека мониторинга запускных файлов

```
// FileMonitor.cpp : Defines the entry point for the DLL application
//
#include <windows.h>
#include "stdafx.h"
#include "FileMonitor.h"

HHOOK SysHook;
HINSTANCE hInst;

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                             )
{
       hInst=(HINSTANCE)hModule;
    return TRUE;
}

LRESULT CALLBACK SysMsgProc(

    int code,       // hook code
    WPARAM wParam,       // removal flag
    LPARAM lParam        // address of structure with message
   )
{
       // Передать сообщение другим ловушкам в системе
       CallNextHookEx(SysHook, code, wParam, lParam);

       if (code == HCBT_ACTIVATE)
       {
              char windtext[255];
              HWND Wnd=((tagMSG*)lParam)->hwnd;
              GetWindowText(Wnd, windtext, 255);

               // (Здесь можно сохранить заголовок активного окна)
       }

       if (code == HCBT_CREATEWND)
       {
```

```
            char windtext[255];
            HWND Wnd=((tagMSG*)lParam)->hwnd;
            GetWindowText(Wnd, windtext, 255);

            // (Здесь можно сохранить заголовок нового окна)
       }
       return 0;
}

///////////////////////////////////////////////////////////////////

DllExport void RunStopHook(bool State, HINSTANCE hInstance)
{
   if (true)
      SysHook = SetWindowsHookEx(WH_CBT, &SysMsgProc, hInst, 0);
   else
      UnhookWindowsHookEx(SysHook);
}
```

Когда создано новое окно или активировано уже существующее, то вызывается наша ловушка с добавленным кодом определения названия окна, которое сгенерировало событие. Дальше вы можете добавить свой код, который будет выполнять необходимые действия (например, сохранять в файле дату и время создания или активации окна). А я, в целях экономии места в книге, опущу этот момент и оставлю на ваше усмотрение, потому что дальнейшие действия зависят от поставленной цели и выходят за рамки темы.

Вот таким нехитрым способом можно получить доступ к сообщениям о событиях, произошедших с окнами, и тем самым контролировать, что происходит с программами на компьютере пользователя.

Примечание

Исходный код библиотеки, описанный в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter3\FileMonitor, а исходный код примера для тестирования этой библиотеки — в каталоге \Demo\Chapter3\FileMonitorTest. Для запуска программы необходимо, чтобы библиотека FileMonitor.dll находилась в каталоге, из которого запускается тестовый пример.

3.9. Управление ярлыками на Рабочем столе

На Рабочем столе ярлыки расположены аналогично строкам в элементе управления `List View`, поэтому ими очень легко управлять. Для этого нужно

найти окно с классом ProgMan. Затем внутри этого окна необходимо получить указатель на элемент управления, содержащий ярлыки.

Все сказанное в виде кода выглядит следующим образом:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
```

Здесь ищем окно с заголовком ProgMan. Хотя вы такое окно не видите, оно существует еще со времен Windows третьей версии (может, и раньше) и называется **Program Manager**. Далее, с помощью функции GetWindow определяется дочернее окно. После этого находим следующее дочернее окно. Вот теперь мы получили указатель на системный объект класса SysListView32. Этот элемент как раз и содержит все ярлыки Рабочего стола.

Мы можем управлять ярлыками, посылая сообщения с помощью функции SendMessage. Например, если выполнить следующую строку кода, то все ярлыки будут упорядочены по левому краю экрана:

```
SendMessage(DesktopHandle, LVM_ARRANGE, LVA_ALIGNLEFT, 0);
```

Рассмотрим каждый из параметров функции SendMessage:

- DesktopHandle — окно, которому надо послать сообщение;
- тип сообщения — LVM_ARRANGE, указывает на необходимость отсортировать иконки;
- первый параметр для сообщения — LVA_ALIGNLEFT, упорядочивает иконки по левому краю;
- второй параметр для сообщения — оставляем нулевым.

Если параметр LVA_ALIGNLEFT заменить на LVA_ALIGNTOP, то ярлыки будут выровнены по верхнему краю окна.

Следующая строка кода удаляет все элементы с Рабочего стола:

```
SendMessage(DesktopHandle, LVM_DELETEALLITEMS, 0, 0);
```

Код похож на тот, что мы уже использовали, только здесь посылается команда LVM_DELETEALLITEMS, которая заставляет удалить все элементы. Попробуйте выполнить эту команду, и весь Рабочий стол очистится. Только удаление происходит не окончательно, и после первой же перезагрузки компьютера все вернется на свои места. Но если в системе запустить невидимую программу, которая будет через определенные промежутки времени очищать ярлыки, то эффект будет впечатляющим.

А теперь самое интересное — перемещение ярлыков по экрану. Для этого можно использовать следующий код:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
for (int i=0; i<200; i++)
   SendMessage(DesktopHandle,
           LVM_SETITEMPOSITION, 0, MAKELPARAM(10, i));
```

Как и в предыдущем примере, ищем элемент управления, содержащий ярлыки. Потом запускается цикл от 0 до 200, в котором посылается сообщение функцией `SendMessage` со следующими параметрами:

- окно, которое должно получить сообщение, — в данном случае это элемент управления с иконками;
- сообщение, которое нужно послать, — `LVM_SETITEMPOSITION` (изменяет позицию иконки);
- индекс иконки, которую надо переместить;
- новая позиция — состоит из двух слов: `x` и `y` позиции элемента. Чтобы правильно разместить числа, мы воспользовались функцией `MAKELPARAM`.

Таким образом, можно как угодно шутить над Рабочим столом Windows. Единственный недостаток описанного примера проявляется в Windows XP, где ярлыки двигаются по экрану не плавно, а скачками. Вот такая уж специфика этой версии Windows. Зато в других вариантах — красота полнейшая, и шутка получается очень интересная.



Исходный код этого примера вы можете найти на компакт-диске в каталоге \Demo\Chapter3\ArrangeIcons.

Что еще можно сделать с ярлыками на Рабочем столе? Да практически все, что можно сделать с элементом управления `List View`. Рассмотрим наиболее любопытные возможности.

3.9.1. Анимация текста

Очень интересного эффекта можно добиться с помощью анимации текста ярлыков. Для этого достаточно знать, как изменить цвет, а анимацию после этого можно сделать любым циклом, который по определенному алгоритму будет изменять цвет текста.

Чтобы изменить цвет текста, нужно послать с помощью функции `SendMessage` сообщение `LVM_SETITEMTEXT`. В качестве третьего параметра указывается 0, а последний параметр — это цвет. Например, чтобы сделать цвет текста черным, нужно выполнить следующий код:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
SendMessage(DesktopHandle, LVM_SETITEMTEXT, 0,
                         (LPARAM) (COLORREF)0);
```

Единственное, что может вызвать трудность, — обновление Рабочего стола, потому что изменения будут видны только после перерисовки экрана.

3.9.2. Обновление иконки

Код, который мы рассматривали в *разд. 3.9.1* для анимации, не эффективен, потому что реально не будет видно движения. Пользователь увидит только начальное и конечное положение ярлыка, а перемещение останется за кадром. Чтобы исправить этот недостаток, нужно после изменения позиции обновлять ее. Для этого нужно послать сообщение `LVM_UPDATE`:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
for (int i=0; i<100; i++)
{
      SendMessage(DesktopHandle, LVM_SETITEMPOSITION,
                0, MAKELPARAM(10, i));
      SendMessage(DesktopHandle, LVM_UPDATE, 0, 0);
      Sleep(10);
}
```

Здесь внутри цикла изменяется позиция нулевой иконки и обновляется ее изображение с помощью сообщения `LVM_UPDATE`. В данном случае третий параметр функции `SendMessage` также указывает на номер обновляемого элемента. Если нужно перерисовать вторую иконку, то код будет выглядеть так:

```
SendMessage(DesktopHandle, LVM_UPDATE, 2, 0);
```

3.10. Использование буфера обмена

Шутить можно над чем угодно, и буфер обмена тут не исключение. Вроде безобидная вещь, а может стать очень мощным инструментом в руках хакера. Главное — творческий подход.

Итак, буфер используется для того, чтобы пользователь мог переносить данные из программы в программу или копировать несколько раз одинаковый

текст. Что ожидает пользователь? Вставляемые данные должны соответствовать скопированным. Вот тут мы можем сделать неожиданный ход.

В Windows есть события, с помощью которых можно отслеживать состояние системного буфера. Это необходимо, чтобы кнопка **Вставить** из буфера обмена была доступна, только когда в буфере есть данные необходимого формата. Можно воспользоваться этими возможностями в своих целях.

Давайте создадим программу, которая будет следить за буфером, а при его изменении — портить содержимое. Создайте новое MFC-приложение (можно на основе диалогового окна) с именем ClipboardChange.

Добавим два новых события, которые должна будет обрабатывать наша программа, чтобы следить за состоянием буфера: `ON_WM_CHANGECBCHAIN` и `ON_WM_DRAWCLIPBOARD`. Для этого откройте файл ClipboardChangeDlg.cpp, найдите карту сообщений и добавьте туда названия необходимых нам событий:

```
BEGIN_MESSAGE_MAP(CClipboardChangeDlg, CDialog)
       ON_WM_CHANGECBCHAIN()
       ON_WM_DRAWCLIPBOARD()
       ON_WM_SYSCOMMAND()
       ON_WM_PAINT()
       ON_WM_QUERYDRAGICON()
       //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Теперь откройте файл ClipboardChangeDlg.h и добавьте в него описания функций, которые будут вызываться в ответ на события буфера обмена. Их нужно объявить в разделе `protected` нашего класса следующим образом:

```
afx_msg void OnChangeCbChain(HWND hWndRemove, HWND hWndAfter);
afx_msg void OnDrawClipboard();
```

Нам также понадобится переменная типа `HWND`, в которой будет храниться указатель на окно-просмотрщик буфера. Назовите ее `ClipboardViewer`.

Снова вернитесь в файл ClipboardChangeDlg.cpp, где нужно добавить код этих функций. Но они не будут вызываться, пока мы не сделаем нашу программу наблюдателем за буфером обмена. Для этого в функции `OnInitDialog` добавьте строку:

```
ClipboardViewer = SetClipboardViewer();
```

Вот теперь можно перейти к рассмотрению двух функций, которые вызываются на события буфера. Код обеих функций приведен в листинге 3.10.

Листинг 3.10. Функции наблюдения за буфером

```
void CClipboardChangeDlg::OnChangeCbChain(HWND hWndRemove, HWND hWndAfter)
{
    if (ClipboardViewer == hWndRemove)
        ClipboardViewer = hWndAfter;

    if (NULL != ClipboardViewer)
    {
        ::SendMessage (ClipboardViewer, WM_CHANGECBCHAIN,
                        (WPARAM) hWndRemove, (LPARAM) hWndAfter);
    }

    CClipboardChangeDlg::OnChangeCbChain(hWndRemove, hWndAfter);
}

void CClipboardChangeDlg::OnDrawClipboard()
{
       if (!OpenClipboard())
       {
              MessageBox("The clipboard is temporarily unavailable");
              return;
       }
       if (!EmptyClipboard())
       {
              CloseClipboard();
              MessageBox("The clipboard cannot be emptied");
              return;
       }

       CString Text="You are hacked";
       HGLOBAL hGlobal = GlobalAlloc(GMEM_MOVEABLE, Text.GetLength()+1);

       if (!hGlobal)
       {
              CloseClipboard();
              MessageBox(CString("Memory allocation error"));
              return;
       }

       strcpy((char *)GlobalLock(hGlobal), Text);
       GlobalUnlock(hGlobal);
       if (!SetClipboardData(CF_TEXT, hGlobal))
```

```
        {
                MessageBox("Error setting clipboard");
        }
        CloseClipboard();
}
```

Самое интересное происходит в функции `OnDrawClipboard`, которая вызывается каждый раз, когда в буфер попадают новые данные. По этому событию надо очищать содержимое буфера обмена и помещать туда свои данные, т. е. пользователь не сможет воспользоваться операцией копирования.

Прежде чем работать с буфером, его необходимо открыть. Для этого используется функция `OpenClipboard`. Если открытие прошло успешно, то она возвращает `TRUE`.

После этого очищается буфер обмена с помощью функции `EmptyClipboard`. Если функция отработала успешно, то она возвращает `TRUE`, иначе буфер закрывается, и выводится сообщение об ошибке. Для закрытия используется функция `CloseClipboard`.

Теперь можно копировать свои данные в буфер обмена. Для этого нужно в глобальной области выделить память необходимого объема и скопировать туда нужный текст. Я поместил туда сообщение "You are hacked". После этого переносим данные из памяти в буфер с помощью функции `SetClipboardData`, у которой есть два параметра:

- константа, определяющая тип данных, — `CF_TEXT`, соответствует текстовым данным;
- указатель на данные, которые должны быть помещены в буфер обмена.

После работы следует обязательно закрыть буфер с помощью функции `CloseClipboard`. Мы хоть и злые, но правильные программисты!

Вот таким простым способом, благодаря нашему воображению, абсолютно безобидный буфер обмена превратился в интересную шутку.

Попробуйте запустить программу и скопировать что-нибудь в буфер обмена. После вставки вместо скопированных данных вы увидите текст "You are hacked".

Примечание

Исходный код программы, описанной в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter3\ClipboardChange.

Глава 4



Работа с сетью

Я напомню, что первоначальный смысл слова "хакер" был больше связан с человеком, который хорошо знает программирование, внутренности ОС и сеть. Именно поэтому достаточно большая часть этой книги посвящена программированию сетевых приложений. В предыдущих главах мы учились понимать внутренности ОС на интересных шуточных примерах. Теперь перейдем к рассмотрению сетевых приложений. На мой взгляд, это не менее интересная тема.

В этой главе я начну знакомить вас с сетевыми возможностями языка программирования C++. Я покажу, как написать множество простых, но эффективных утилит с помощью объектов Visual C++ и сетевой библиотеки WinSock.

Для начала, я ограничусь использованием объектной модели, которую предоставляет среда разработки, а вот чуть позже мы познакомимся с низкоуровневым программированием сетей. Но это чуть позже.

Я не захотел сразу загружать вас низкоуровневым программированием с использованием API-функций, потому что может получиться переполнение мозгового буфера. Уж лучше мы будем все делать постепенно. Сначала познакомимся с простыми вещами, не заглядывая в дебри, а потом перейдем к более сложному материалу.

4.1. Теория сетей и сетевых протоколов

Прежде чем я покажу первый пример, придется немного заняться теорией. Это не займет много времени, но потом нам будет легче понимать друг друга. Для лучшего восприятия материала этой главы вам желательно знать основы сетей и протоколов.

Каждый раз, когда вы передаете данные по сети, они как-то перетекают от вашего компьютера к серверу или другому компьютеру. Как это происходит? Вы, наверно, скажете, что с помощью специального сетевого протокола, и будете правы. Но существует множество разновидностей протоколов. Какой и когда используется? Зачем они нужны? Как они работают? Вот на эти вопросы я сейчас постараюсь дать ответ.

Прежде чем разбираться с протоколами, нам необходимо узнать, что такое модель взаимодействия открытых систем (OSI — Open Systems Interconnection), которая была разработана Международной организацией по стандартам (ISO — International Organization for Standardization). В соответствии с этой моделью, сетевое взаимодействие делится на семь уровней.

1. *Физический уровень* — передача битов по физическим каналам (коаксиальный кабель, витая пара, оптоволоконный кабель). Здесь определяются характеристики физических сред и параметры электрических сигналов.
2. *Канальный уровень* — передача кадра данных между любыми узлами в сетях типовой топологии или соседними узлами в сетях произвольной топологии. В качестве адресов на канальном уровне используются физические (MAC Media Access Control address) адреса. Это 48-битовое число, которое позволяет однозначно идентифицировать устройство в сети. Теоретически такой адрес должен быть жестко прописан в устройстве, но практически его легко менять с помощью средств ОС.
3. *Сетевой уровень* — доставка пакета любому узлу в сетях произвольной топологии. На этом уровне нет никаких гарантий доставки пакета.
4. *Транспортный уровень* — доставка пакета любому узлу с любой топологией сети и заданным уровнем надежности доставки. На этом уровне имеются средства для установления соединения, буферизации, нумерации и упорядочивания пакетов.
5. *Уровень сеанса* — управление диалогом между узлами. Обеспечена возможность фиксации активной на данный момент стороны.
6. *Уровень представления* — предоставляется возможность преобразования данных (шифрование, сжатие).
7. *Прикладной уровень* — набор сетевых сервисов (FTP, E-mail и др.) для пользователя и приложения.

Если вы внимательно прочитали обо всех уровнях, то, наверно, заметили, что первые три уровня обеспечиваются оборудованием, таким как сетевые карты, маршрутизаторы, концентраторы, мосты и др. Последние три — операционной системой или приложением. Четвертый уровень является промежуточным.

Как работает протокол по этой модели? Все начинается с прикладного уровня. Пакет попадает на этот уровень, и к нему добавляется заголовок. После этого прикладной уровень отправляет этот пакет на следующий уровень (уровень представления). Здесь ему также добавляется свой собственный заголовок, и пакет отправляется дальше. И так до физического уровня, который занимается непосредственно передачей данных и отправляет пакет в сеть.

Другая машина, получив пакет, начинает обратный отсчет. Пакет с физического уровня попадает на канальный. Канальный уровень убирает свой заголовок и поднимает пакет выше (на уровень сети). Уровень сети убирает свой заголовок и поднимает пакет выше. Так пакет поднимается до уровня приложения, где остается чистый пакет без служебной информации, которая была прикреплена на исходном компьютере перед отправкой данных.

Передача данных не обязательно должна начинаться с седьмого уровня. Если используемый протокол работает на четвертом уровне, то процесс передачи начнется с него, и пакет будет подниматься вверх до физического уровня для отправки. Количество уровней в протоколе определяет его потребности и возможности при передаче данных.

Чем ниже находится протокол (ближе к прикладному уровню), тем больше у него возможностей и больше накладных расходов при передаче данных (длиннее и сложнее заголовок). Рассматриваемые в данной книге протоколы будут находиться на разных уровнях, поэтому будут иметь разные возможности.

Корпорация Microsoft реализовала протокол TCP/IP в модели OSI по-своему (с небольшими отклонениями от стандарта). Модель OSI справочная и предназначена только в качестве рекомендации, поэтому ее изменили почти до неузнаваемости.

У MS TCP/IP вместо семи уровней есть только четыре. Но это не значит, что остальные уровни позабыты и позаброшены, просто один уровень может выполнять все, что в OSI делают три уровня. Например, уровень приложения у Microsoft выполняет все, что делают уровень приложения, уровень представления и уровень сеанса, вместе взятые.

На рис. 4.1 схематично сопоставлены MS TCP/IP-модель и справочная модель OSI. Слева указаны названия уровней по методу MS, а справа — уровни OSI. В центре показаны протоколы. Я постарался разместить их именно на том уровне, на котором они работают, впоследствии нам это пригодится.

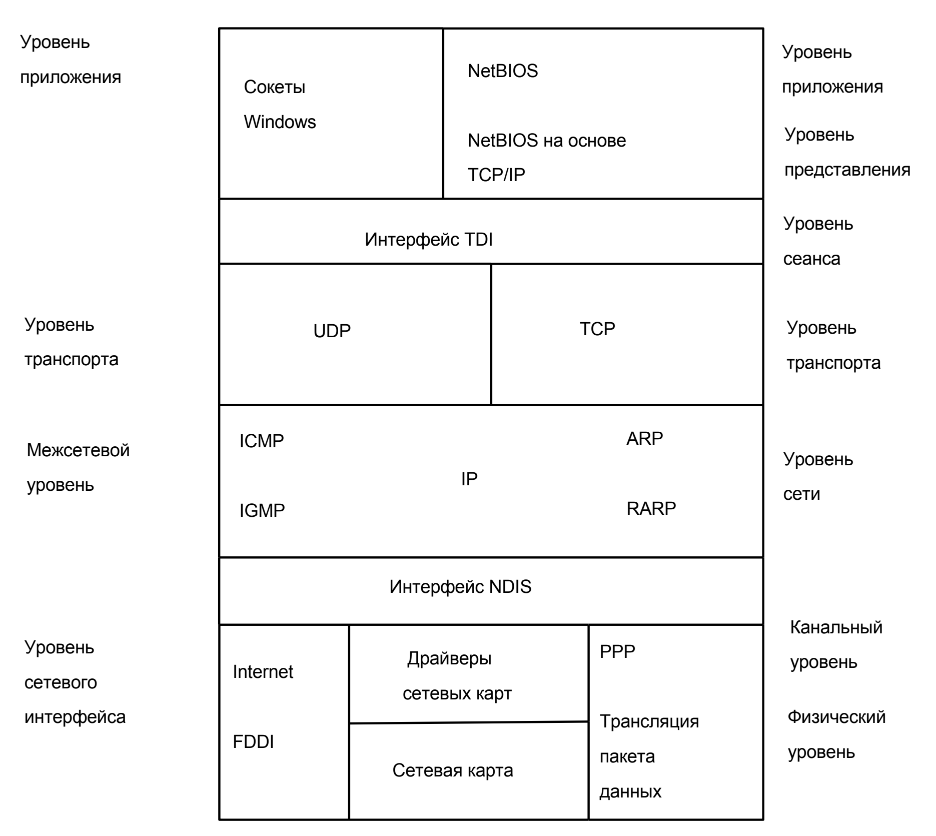


Рис. 4.1. Модель OSI и вариант от MS

4.1.1. Сетевые протоколы

Прежде чем начинать писать сетевые программы, необходимо разобраться с сетевыми протоколами, понять основу и принципы их работы. В этом разделе я остановлюсь на самых важных моментах, которые необходимо знать программисту для правильного принятия решения. Вы увидите основные различия и сможете понять, что нельзя просто взять первый попавшийся протокол и написать с его помощью любую программу. Иногда выбор бывает очень сложным, но от него зависит будущее программы.

Протокол IP

Если посмотреть на схему сетевой модели (см. рис. 4.1), то можно увидеть, что протокол IP находится на сетевом уровне. Из этого можно сделать вывод,

что IP выполняет сетевые функции — доставка пакета любому узлу в сетях произвольной топологии.

Протокол IP при передаче данных не устанавливает виртуального соединения и использует датаграммы (пакеты данных) для отправки информации от одного компьютера к другому. Это значит, что по протоколу IP пакеты просто отправляются в сеть без ожидания подтверждения о получении данных, а значит, без гарантии доставки пакетов и соответственно без гарантии целостности данных. Когда нужно отправить большой файл, то он должен быть разбит на части, например, на 100 частей. Если хотя бы один пакет из 100 необходимых не дойдет до адресата, то данные нарушатся, и собрать их в единое целое будет невозможно.

Все необходимые действия по подтверждению и обеспечению целостности данных должны обеспечивать протоколы, работающие на более высоком уровне.

Каждый IP-пакет содержит адреса отправителя и получателя, идентификатор протокола, TTL (время жизни пакета) и контрольную сумму для проверки целостности пакета. Как видите, здесь есть контрольная сумма, которая все же позволяет узнать целостность пакета. Но об этом узнает только получатель. Когда компьютер-получатель принял пакет, то он проверяет контрольную сумму только для себя. Если сумма сходится, то пакет обрабатывается, иначе просто игнорируется. А компьютер-отправитель не сможет узнать об ошибке, возникшей в пакете, и повторить посылку.

Еще один интересный вопрос: а что если первый пакет дойдет до получателя неверным? Отправитель не будет этого знать и продолжит бессмысленную отправку всех остальных пакетов. Так что соединение по протоколу IP нельзя считать надежным.

Сопоставление адреса ARP и RARP

Протокол ARP (Address Resolution Protocol, протокол определения адреса) предназначен для определения аппаратного (MAC) адреса компьютера в сети по его IP-адресу. Прежде чем данные могут быть посланы на какой-нибудь компьютер по локальной сети, отправитель должен знать аппаратный адрес получателя. Именно для этого и предназначен ARP.

Когда компьютер посылает ARP-запрос на поиск аппаратного адреса, то протокол сначала ищет этот адрес в локальном кэше. Если уже были обращения по данному IP-адресу, то информация о MAC-адресе должна сохраниться в кэше. Если ничего не найдено, то в сеть посылается широковещательный запрос с вопросом, чей это адрес, который получат все компьютеры сети. Они получат этот пакет и проверят адрес. Тот, кому принадлежит искомый IP, ответит на запрос, указав свой MAC-адрес. Так как этот адрес должен быть уникальным (прошивается в сетевом устройстве на заводе-изготовителе), то и

ответ должен быть один. Но вы должны учитывать, что есть средства подделки MAC-адресов (хакеры иногда используют этот прием в своих целях), и может возникнуть ситуация, когда ответ придет от двух машин.

Протокол RARP (Revers Address Resolution Protocol, обратный протокол определения адреса) определяет IP-адрес по известному MAC-адресу. Процесс поиска адресов абсолютно такой же.

4.1.2. Транспортные протоколы

На транспортном уровне мы имеем два протокола: UDP и TCP. Оба они работают поверх IP. Это значит, что, когда пакет TCP или UDP опускается на уровень ниже для отправки в сеть, он попадает на уровень сети прямо в лапы протокола IP. Здесь пакету добавляется сетевой адрес, TTL и другие атрибуты протокола IP. После этого пакет идет дальше вниз для физической отправки в сеть. "Голый" пакет TCP не может быть отправлен в сеть, потому что он не имеет информации о получателе, эта информация добавляется к пакету с IP-заголовком на уровне сети.

Давайте теперь рассмотрим каждый протокол в отдельности более подробно.

Быстрый UDP

Как и IP, протокол UDP для передачи данных не устанавливает соединения с сервером. Данные просто выбрасываются в сеть, и протокол даже не заботится о доставке пакета. Если данные на пути к серверу испортятся или вообще не дойдут, то отправляющая сторона об этом не узнает. Так что по этому протоколу не следует передавать очень важные данные.

Благодаря тому, что протокол UDP не устанавливает соединения, он работает очень быстро (в несколько раз быстрее TCP, о котором чуть позже). Из-за высокой скорости его удобно использовать там, где не нужно заботиться о целостности данных. Таким примером могут служить радиостанции в Интернете. Звуковые данные просто выплескиваются в глобальную сеть, и если слушатель не получит одного пакета, то максимум, что он заметит — небольшое заикание в месте потери. Но если учесть, что сетевые пакеты имеют небольшой размер, то эта задержка будет практически незаметна.

Большая скорость — большие проблемы с безопасностью. Так как нет соединения между сервером и клиентом, то нет никакой гарантии в достоверности данных. Протокол UDP больше подвержен спуфингу (spoofing, подмена адреса отправителя), поэтому построение на нем защищенных сетей затруднено.

Итак, UDP очень быстр, но его можно использовать только там, где данные не имеют высокой ценности (возможна потеря отдельных пакетов) и не секретны (UDP больше подвержен взлому).

Медленный, но надежный TCP

Как я уже сказал, протокол TCP лежит на одном уровне с UDP и работает поверх IP, который используется для отправки данных. Именно поэтому протоколы TCP и IP неразрывно связаны и их часто объединяют одним названием TCP/IP.

В отличие от UDP-протокол TCP устраняет недостатки своего транспорта (IP). В этом протоколе заложены средства установления связи между приемником и передатчиком, обеспечение целостности данных и гарантии их доставки. Прежде чем отправлять данные, одна из сторон должна стать сервером и запустить ожидание соединения, а другая сторона должна выступать в качестве клиента. Клиент устанавливает соединение с сервером и после этого может начинаться обмен данными. Этот обмен происходит, пока соединение не будет разорвано.

Когда данные отправляются в сеть по TCP, то на отправляющей стороне включается таймер. Если в течение определенного времени приемник не подтвердит получение данных, то будет предпринята еще одна попытки отправки данных. Если приемник получит испорченные данные, то он сообщит об этом источнику и попросит снова отправить испорченные пакеты. Благодаря этому обеспечивается гарантированная доставка данных.

Когда нужно отправить сразу большую порцию данных, не вмещающихся в один пакет, то они разбиваются на несколько TCP-пакетов. Пакеты отправляются порциями по несколько штук (зависит от настроек стека). Когда сервер получает порцию пакетов, то он восстанавливает их очередность и собирает данные вместе (даже если пакеты прибыли не в том порядке, в котором они отправлялись).

Из-за лишних накладных расходов на установку соединения подтверждение доставки и повторную пересылку испорченных данных протокол TCP намного медленней UDP. Зато TCP можно использовать там, где нужна гарантия доставки и большая надежность. Хотя надежность нельзя назвать сильной (нет шифрования, сохраняется возможность взлома), но она приемлемая и намного больше, чем у UDP. По крайней мере, тут спуфинг не может быть реализован так просто, как у UDP, и в этом вы убедитесь, когда прочтете про процесс установки соединения. Хотя возможно все, и хакеры умеют взламывать и TCP-протокол.

Опасные связи TCP

Давайте посмотрим, как протокол TCP обеспечивает надежность соединения. Все начинается еще на этапе попытки соединения двух компьютеров в следующей последовательности:

1. Клиент, который хочет соединиться с сервером, отправляет SYN-запрос на сервер, указывая номер порта, к которому он хочет подсоединиться, и специальное число (чаще всего случайное).
2. Сервер отвечает своим сегментом SYN, содержащим специальное число сервера. Он также подтверждает приход SYN-пакета со стороны клиента с использованием ACK-ответа, где специальное число, отправленное клиентом, увеличено на 1.
3. Клиент должен подтвердить приход SYN от сервера с использованием ACK — специальное число сервера плюс 1.

Получается, что при соединении клиента с сервером они обмениваются специальными числами. Эти числа и используются в дальнейшем для обеспечения целостности и защищенности связи. Если кто-то другой захочет вклиниться в установленную связь (с помощью спуфинга), то ему надо будет подделать эти числа. Но т. к. они большие и выбираются случайным образом, то такая задача достаточно сложная.



Кевин Митник в свое время смог решить ее. Но это уже другая история, и не будем уходить далеко в сторону.

Стоит еще отметить, что приход любого пакета подтверждается ACK-ответом, что гарантирует доставку данных.

4.1.3. Прикладные протоколы — загадочный NetBIOS

NetBIOS (Network Basic Input Output System, базовая система сетевого ввода/вывода) — это стандартный интерфейс прикладного программирования. А проще говоря, это всего лишь набор API-функций для работы с сетью (хотя весь NetBIOS состоит только из одной функции, но зато какой!). NetBIOS был разработан в 1983 г. компанией Sytek Corporation специально для IBM.

Система NetBIOS определяет только программную часть передачи данных, т. е. как должна работать программа для передачи данных по сети. А вот как будут физически передаваться данные, в этом документе не говорится ни слова, да и в реализации отсутствует что-нибудь подобное.

Если посмотреть на рис. 4.1, то можно увидеть, что NetBIOS находится в самом верху схемы. Он расположен на уровнях сеанса, представления и приложения. Такое его расположение — лишнее подтверждение моих слов.

NetBIOS только формирует данные для передачи, а физически передаваться они могут только с помощью других протоколов, например, TCP/IP, IPX/SPX и т. д. Это значит, что NetBIOS является независимым от транспорта. Если

другие варианты протоколов верхнего уровня (только формирующие пакеты, но не передающие) привязаны к определенному транспортному протоколу, который должен передавать сформированные данные, то пакеты NetBIOS может передавать любой другой протокол. Прочувствовали силу? Представьте, что вы написали сетевую программу, работающую через NetBIOS. А если вы еще не знаете, то она будет прекрасно работать как в UNIX/Windows-сетях через TCP, так и в Novell-сетях через IPX.

С другой стороны, для того чтобы два компьютера смогли соединиться друг с другом по NetBIOS, необходимо, чтобы на обоих стоял хотя бы один общий транспортный протокол. Если один компьютер будет посылать NetBIOS-пакеты через TCP, а другой — с помощью IPX, то эти компьютеры друг друга не поймут. Транспорт должен быть одинаковый.

Стоит сразу же отметить, что не все варианты транспортных протоколов по умолчанию могут передавать по сети NetBIOS-пакеты. Например, IPX/SPX сам по себе этого не умеет. Чтобы его обучить, нужно иметь "NWLink IPX/SPX/NetBIOS Compatible Transport Protocol".

Так как NetBIOS чаще всего использует в качестве транспорта протокол TCP, который работает с установкой виртуального соединения между клиентом и сервером, то по этому протоколу можно передавать достаточно важные данные. Целостность и надежность передачи будет осуществлять TCP/IP, а NetBIOS дает только удобную среду для работы с пакетами и программирования сетевых приложений. Так что если вам нужно отправить в сеть какие-либо файлы, то можно смело положиться на NetBIOS.

С другой стороны, данный протокол достаточно сложен и неудобен в разработке, поэтому с каждым годом все больше и больше теряет в популярности. В данной книге я не буду уделять ему много внимания.

4.1.4. NetBEUI

В 1985 г. уже IBM сделала попытку превратить NetBIOS в полноценный протокол, который умеет не только формировать данные для передачи, но и физически передавать их по сети. Для этого был разработан NetBEUI (NetBIOS Extended User Interface, расширенный пользовательский интерфейс NetBIOS). Он предназначен именно для описания физической части передачи данных протокола NetBIOS.

Сразу хочу отметить, что NetBEUI является немаршрутизируемым протоколом, и первый же маршрутизатор будет отбиваться от таких пакетов, как Мария Шарапова от теннисных мячиков. Это значит, что если между двумя компьютерами стоит маршрутизатор, и нет другого пути для связи, то им не удастся установить соединение через NetBEUI.

4.1.5. Сокеты Windows

Сокеты (Sockets) — это всего лишь программный интерфейс, который облегчает взаимодействие между различными приложениями. Современные сокеты родились из программного сетевого интерфейса, реализованного в ОС BSD UNIX. Тогда этот интерфейс создавался для облегчения работы с TCP/IP на верхнем уровне.

С помощью сокетов легко реализовать большинство известных протоколов, которые используются каждый день при выходе в Интернет. Достаточно только назвать HTTP, FTP, POP3, SMTP и далее в том же духе. Все они используют для отправки своих данных TCP или UDP и легко программируются с помощью библиотеки sockets/winsock.

4.1.6. Протоколы IPX/SPX

Осталось только рассказать еще о нескольких протоколах, которые встречаются в повседневной жизни чуть реже, но зато они не менее полезны. Первые на очереди — это IPX/SPX.

Протокол IPX (Internetwork Packet Exchange, межсетевой обмен пакетами) раньше использовался в сетях фирмы Novell. В наших любимых "окошках" есть специальная служба **Клиент для сетей Novell**, с помощью которой вы сможете работать в таких сетях. IPX работает подобно IP и UDP: без установления связи, а значит, без гарантии доставки и всех последующих достоинств и недостатков.

SPX (Sequence Packet Exchange, последовательный обмен пакетами) — это транспорт для IPX, который работает с установлением связи и обеспечивает целостность данных. Так что если вам понадобится надежность при использовании IPX, то используйте связку IPX/SPX или IPX/SPX11.

Сейчас IPX уже теряет свою популярность, но еще помнятся времена DOS, когда все сетевые игры работали через этот протокол.

Как видите, в Интернете протоколов целое море, но большинство из них взаимосвязано, как, например, HTTP/TCP/IP. Протокол, предназначенный для одной цели, может оказаться абсолютно непригодным для другой, потому что создать что-то идеальное невозможно. У каждого будут свои достоинства и недостатки.

И все же модель OSI, принятая еще на заре Интернета, не утратила своей актуальности до сих пор. По ней работает все и вся. Главное ее достоинство — скрывать сложность сетевого общения между компьютерами, с чем OSI справляется без особых проблем.

4.1.7. Сетевые порты

Прежде чем вы начнете писать собственные программы, надо разобраться с еще одним понятием — *сетевой порт*. Допустим, что вашему компьютеру на сетевую карту пришел пакет данных. Как операционная система должна определить, для какой программы пришли данные: для Internet Explorer, для почтового клиента или для вашей программы? Чтобы определить это, используются порты.

Когда программа соединяется с сервером, то она открывает на вашем компьютере какой-нибудь сетевой порт и сообщает серверу, что именно с этим портом она работает. После этого сервер будет посылать на ваш компьютер пакеты данных, в которых будет указан сетевой адрес компьютера и номер порта. По IP-адресу пакет будет доставлен до вашего компьютера, а по номеру порта операционная система определит, что именно для вашей программы предназначается пришедший пакет.

Для соединения с сервером вам надо знать не только IP-адрес сервера, но и порт, на котором работает программа, потому что на сервере может работать множество сетевых программ, и все они используют свои порты.

Из всего вышесказанного следует, что только одна программа может открыть определенный порт. Если бы две программы могли открывать, например, порт 21, то операционная система уже не смогла бы определить, от какой из двух программ пришли данные. На самом деле, есть способ двум программам работать с одним портом, но эту ситуацию пока опустим.

Номер порта — это число от 1 до 65 535. Для передачи такого числа по сети достаточно всего лишь двух байтов, поэтому это не будет накладно для сети. Я рекомендую использовать для своих целей порты с номерами более 1024, потому что среди меньших значений очень много зарегистрированных номеров, и у вашей программы увеличивается вероятность конфликта с другими сетевыми программами.

Теперь пора переходить к более подробному рассмотрению некоторых протоколов и сетевых возможностей Windows. Я не смогу объяснить абсолютно все, но постараюсь рассмотреть самое интересное в сетевом программировании и показать несколько полезных примеров.

4.2. Работа с ресурсами сетевого окружения

В ОС Windows есть очень удобная возможность — обмениваться информацией между компьютерами через открытые ресурсы. Вы можете сделать какую-либо папку открытой для сетевого доступа, и любой пользователь в вашей сети, у которого есть соответствующие права, сможет обращаться к фай-

лам этой папки. Можно также подключить открытую папку как локальный диск. В любом случае для доступа к таким ресурсам можно использовать стандартные функции для доступа к файлам.

Когда приложение использует файл, то ОС сначала определяет устройство, на котором находится необходимый ресурс. Если ресурс расположен на удаленном компьютере, то запрос на ввод/вывод передается по сети этому устройству. Таким образом, ОС при обращении к сетевому ресурсу занимается перенаправлением ввода/вывода (I/O redirection).

Допустим, что у вас диск Z: — это подключенная по сети папка удаленного компьютера. Каждый раз, когда вы обращаетесь к ней, ОС переадресует запросы ввода/вывода перенаправителю (redirector), который создаст сетевой канал связи с удаленным компьютером для доступа к его ресурсам. Таким образом, можно использовать те же средства, что и для доступа к локальным ресурсам. Это сильно облегчает создание приложений, предназначенных для работы в локальной сети. Точнее сказать, никаких изменений вносить не надо. Если программа умеет работать с локальным диском, то сможет работать и с удаленными дисковыми ресурсами, потому что для прикладной программы весь процесс становится прозрачным.

Для более подробной информации по работе редиректора можете обратиться к документации по Windows или специализированной литературе. Для простого пользователя, и даже программиста, эта информация не очень важна, потому что весь процесс перенаправления скрыт.

Чтобы обеспечить доступ к ресурсам другого компьютера в вашей сети, не обязательно подключать открытую папку как локальный диск. Достаточно правильно указать сетевой путь. Для этого надо знать универсальные правила именования (Universal Naming Conversion, UNC) — способ доступа к файлам и устройствам (например, к принтерам) без назначения им буквы локального диска. Тогда вы не будете зависеть от имен дисков, но нужно будет четко определить имя компьютера, на котором находится нужный объект.

Общий вид UNC-имени выглядит следующим образом:

```
\\компьютер\имя\путь
```

Имя начинается с двойной косой черты (`\\`). Затем идет имя компьютера или сервера, на котором расположен объект. `Имя` — это имя сетевой папки. После него нужно указать путь к объекту.

Допустим, что у вас есть компьютер Tom, на котором открыта для общего доступа папка Sound. В этой папке есть файл MySound.wav. Для доступа к этому файлу необходимо использовать UNC-имя: `\\Tom\Sound\MySound.wav`.

В листинге 4.1 приведен пример создания файла в открытой папке компьютера с именем Notebook.

Листинг 4.1. Пример создания файла в открытой папке другого компьютера

```
void CreateNetFile()
{
    HANDLE FileHandle;
    DWORD BWritten;

    // Create file \\notebook\temp\myfile.txt
    // (Создание файла \\notebook\temp\myfile.txt

    if ((FileHandle = CreateFile("\\\\notebook\\temp\\myfile.txt",
              GENERIC_WRITE | GENERIC_READ,
              FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
              CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
            )) == INVALID_HANDLE_VALUE)
    {
        MessageBox(0, "Create file error", "Error",0);
        return;
    }

    // Write to file 9 symbols
    // (Записать в файл 9 символов)
    if (WriteFile(FileHandle, "Test line", 9, &BWritten, NULL)== 0)
    {
        MessageBox(0, "Write to file error", "Error",0);
        return;
    }

    // Close file  (Закрыть файл)
    CloseHandle(FileHandle);
}
```

Для начала создается файл с помощью стандартной WinAPI-функции `CreateFile`. У этой функции следующие параметры:

- путь к создаваемому файлу;
- режим доступа — файл открыт для чтения (`GENERIC_READ`) и записи (`GENERIC_WRITE`);
- режим доступа к открытому файлу другим программам — другим приложениям разрешено чтение (`FILE_SHARE_READ`) и запись (`FILE_SHARE_WRITE`);
- атрибуты безопасности — не использованы (`NULL`);
- способ открытия файла — всегда создавать (`CREATE_ALWAYS`), если файл уже существует, то данные будут перезаписаны;

- атрибуты создаваемого файла — нормальное состояние файла (FILE_ATTRIBUTE_NORMAL);
- указатель на шаблон, который будет использоваться при создании файла.

Функция CreateFile возвращает указатель на открытый файл. Если результат равен INVALID_HANDLE_VALUE, то файл не был создан по каким-либо причинам.

Для записи используется функция WriteFile, у которой следующие параметры:

- указатель на открытый файл;
- данные, которые надо записать;
- количество байтов данных для записи;
- количество записанных байтов (переменную типа DWORD);
- структура, которая необходима только при открытии файла в режиме наложения (overlapped I/O).

Если запись прошла успешно, то функция должна вернуть ненулевое значение.

После всех манипуляций с файлом его необходимо закрыть. Для этого вызывается функция CloseHandle, которой нужно только передать указатель на файл, который надо закрыть.

Вот и все!!! Никаких сетевых соединений, никаких портов. За нас все делает ОС Windows. В этом отношении она спроектирована достаточно удобно, как для пользователей, так и для программистов.



Исходный код примера вы можете найти на компакт-диске в каталоге \Demo\Chapter4\Network.

4.3. Структура сети

Для того чтобы просмотреть доступные в вашей сети компьютеры, нужно воспользоваться сетевым окружением. Но что, если вам нужно в своей программе сделать просмотр сети? Это очень просто. Сейчас я продемонстрирую программу, с помощью которой можно будет в виде дерева увидеть все компьютеры в сети и их открытые ресурсы.

Создайте новое MFC-приложение в Visual C++ и назовите проект NetNeighbour. В мастере создания приложений, в разделе **Application Type** выберите **Dialog based**, а в разделе **Advanced Features** — **Windows sockets**.

Нажмите кнопку **Finish**, чтобы среда разработки создала необходимый шаблон приложения.

Прежде чем приступать к программированию, необходимо оформить окно будущей программы. Откройте в редакторе ресурсов диалоговое окно `IDD_NETNEIGHBOUR_DIALOG`. Растяните по всей свободной поверхности компонент `Tree Control` (рис. 4.2).

Чтобы можно было работать с этим компонентом, щелкните по нему правой кнопкой мыши. В появившемся меню выберите пункт **Add variable**, а в поле **Variable name** укажите `m_NetTree`. Эта переменная понадобится для добавления в меню новых пунктов.

Теперь все готово для рассмотрения исходного кода. Перейдите в файл NetNeighbourDlg.cpp. Здесь найдите функцию `OnInitDialog`, которая вызывается во время инициализации окна. В этот момент необходимо создать корневой элемент дерева. Это должно происходить следующим образом:

```
m_hNetworkRoot = InsertTreeItem(TVI_ROOT, NULL, "My Net",
      DRIVE_RAMDISK+1);
```

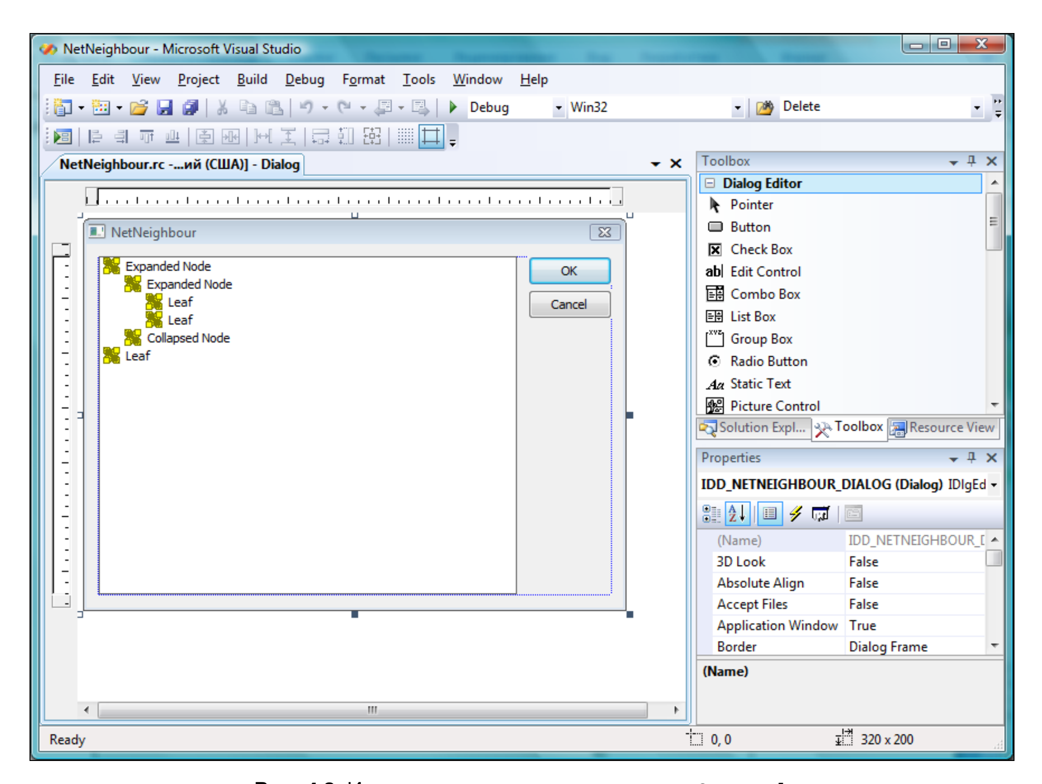


Рис. 4.2. Использование компонента `Tree Control`

В переменной `m_hNetworkRoot` сохраняется результат работы функции `InsertTreeItem`.

Придется несколько раз использовать такой же код для добавления элементов, и чтобы в одном модуле не повторять одни и те же действия, я все оформил отдельной функцией (листинг 4.2).

Листинг 4.2. Добавление нового элемента в дерево сети

```
HTREEITEM CNetNeighbourDlg::InsertTreeItem(HTREEITEM hParent,
       NETRESOURCE *const pNetResource, CString sText, int iImage)
{
       TVINSERTSTRUCT InsertStruct;
       InsertStruct.hParent                = hParent;
       InsertStruct.hInsertAfter           = TVI_LAST;
       InsertStruct.itemex.mask            = TVIF_IMAGE | TVIF_TEXT |
               TVIF_CHILDREN | TVIF_PARAM;
       InsertStruct.itemex.pszText         =
              sText.GetBuffer(sText.GetLength());
       InsertStruct.itemex.iImage          = iImage;
       InsertStruct.itemex.children        = 1;
       InsertStruct.itemex.lParam          = (LPARAM)pNetResource;
       sText.ReleaseBuffer();
       return m_NetTree.InsertItem( &InsertStruct );
}
```

Теперь программа выглядит должным образом и создает корневой элемент, но пока без поиска в сети. Когда программа запущена, и пользователь щелкнет мышью по элементу дерева, нужно найти все, что есть доступного в сети, относящееся к этому элементу.

Для этого надо написать обработчик события `ITEMEXPANDING` и в нем производить поиск. Перейдите в редактор ресурсов и выделите компонент `Tree Control`. В окне **Properties** щелкните по кнопке **Control Events**, и вы увидите все события, которые может генерировать выделенный компонент. Щелкните напротив события `TVN_ITEMEXPANDING` и в выпадающем списке выберите пункт **Add**, чтобы добавить обработчик события. Код, который должен быть в этом обработчике, приведен в листинге 4.3.

Листинг 4.3. Обработчик события `TVN_ITEMEXPANDING`

```
void CNetNeighbourDlg::OnTvnItemexpandingTree1(NMHDR *pNMHDR,
      LRESULT *pResult)
```

```
{
   LPNMTREEVIEW pNMTreeView = reinterpret_cast<LPNMTREEVIEW>(pNMHDR);
   // TODO: Add your control notification handler code here
   // (Добавьте сюда ваш код обработки сообщения)

   CWaitCursor CursorWaiting;
   ASSERT(pNMTreeView);
   ASSERT(pResult);

   if (pNMTreeView->action == 2)
   {
      CString sPath = GetItemPath(pNMTreeView->itemNew.hItem);

      if( !m_NetTree.GetChildItem(pNMTreeView->itemNew.hItem))
      {
          EnumNetwork(pNMTreeView->itemNew.hItem);
          If (m_NetTree.GetSelectedItem() != pNMTreeView->itemNew.hItem)
                m_NetTree.SelectItem(pNMTreeView->itemNew.hItem);
      }
   }

   *pResult = 0;
}
```

Здесь у элемента, который в данный момент пытаются открыть, проверяется наличие дочерних элементов и организуется их поиск. Для этого вызывается функция `EnumNetwork`, которую можно увидеть в листинге 4.4.

Листинг 4.4. Функция `EnumNetwork` для просмотра сети

```
bool CNetNeighbourDlg::EnumNetwork(HTREEITEM hParent)
{
   bool bGotChildren = false;

   NETRESOURCE *const pNetResource =
          (NETRESOURCE *)(m_NetTree.GetItemData(hParent));

   DWORD dwResult;
   HANDLE hEnum;
   DWORD cbBuffer = 16384;
   DWORD cEntries = 0xFFFFFFFF;
   LPNETRESOURCE lpnrDrv;
   DWORD i;
```

```
dwResult = WNetOpenEnum(pNetResource ? RESOURCE_GLOBALNET :
                                RESOURCE_CONTEXT,
           RESOURCETYPE_ANY, 0,
           pNetResource ? pNetResource: NULL,
           &hEnum);

if (dwResult != NO_ERROR)
    return false;

do
{
    lpnrDrv = (LPNETRESOURCE) GlobalAlloc(GPTR, cbBuffer);
    dwResult = WNetEnumResource(hEnum, &cEntries, lpnrDrv,
                                               &cbBuffer);
    if (dwResult == NO_ERROR)
    {
        for(i = 0; i<cEntries; i++)
        {
             CString sNameRemote=lpnrDrv[i].lpRemoteName;
             int nType = 9;
             if(sNameRemote.IsEmpty())
             {
                 sNameRemote = lpnrDrv[i].lpComment;
                 nType = 8;
             }
             if (sNameRemote.GetLength() > 0 &&
                                sNameRemote[0] == _T('\\'))
                  sNameRemote = sNameRemote.Mid(1);
             if (sNameRemote.GetLength() > 0 &&
                                sNameRemote[0] == _T('\\'))
                  sNameRemote = sNameRemote.Mid(1);

              if (lpnrDrv[i].dwDisplayType ==
                              RESOURCEDISPLAYTYPE_SHARE)
              {
                  int nPos = sNameRemote.Find( _T('\\'));
                  if (nPos >= 0)
                      sNameRemote = sNameRemote.Mid(nPos+1);
                  InsertTreeItem(hParent, NULL,
                  sNameRemote, DRIVE_NO_ROOT_DIR);
              }
              else
              {
                  NETRESOURCE* pResource    = new NETRESOURCE;
                  ASSERT(pResource);
```

```
                        *pResource = lpnrDrv[i];
                        pResource->lpLocalName    =
                                  MakeDynamic(pResource->lpLocalName);
                        pResource->lpRemoteName =
                                  MakeDynamic(pResource->lpRemoteName);
                        pResource->lpComment       =
                                  MakeDynamic(pResource->lpComment);
                        pResource->lpProvider       =
                                  MakeDynamic(pResource->lpProvider);
                        InsertTreeItem(hParent, pResource,
                                  sNameRemote, pResource->dwDisplayType+7);
                    }
                    bGotChildren = true;
               }
          }
          GlobalFree((HGLOBAL)lpnrDrv);
          if (dwResult != ERROR_NO_MORE_ITEMS)
              break;
      }
      while (dwResult != ERROR_NO_MORE_ITEMS);

      WNetCloseEnum(hEnum);
      return bGotChildren;
}
```

Логика поиска сетевых ресурсов достаточно проста. Для начала нужно открыть поиск функцией `WNetOpenEnum`, которая выглядит следующим образом:

```
DWORD WNetOpenEnum(
    DWORD dwScope,                // Область перечисления
    DWORD dwType,                 // Типы ресурсов для перечисления
    DWORD dwUsage,                // по типу использования
    LPNETRESOURCE lpNetResource, // Указатель на структуру ресурса
    LPHANDLE lphEnum              // Буфер для дескриптора перечисления
   );
```

Функция открывает перечисление сетевых устройств в локальной сети. Рассмотрим передаваемые ей параметры:

❒ `dwScope` — ресурсы, включаемые в перечисление. Возможны комбинации следующих значений:

- `RESOURCE_GLOBALNET` — все ресурсы сети;
- `RESOURCE_CONNECTED` — подключенные ресурсы;
- `RESOURCE_REMEMBERED` — запомненные ресурсы;

- dwType — тип ресурсов, включаемых в перечисление. Возможны комбинации следующих значений:
  - RESOURCETYPE_ANY — все ресурсы сети;
  - RESOURCETYPE_DISK — сетевые диски;
  - RESOURCETYPE_PRINT — сетевые принтеры;
- dwUsage — использование ресурсов, включаемых в перечисления. Возможны следующие значения:
  - 0 — все ресурсы сети;
  - RESOURCEUSAGE_CONNECTABLE — подключаемые;
  - RESOURCEUSAGE_CONTAINER — контейнерные;
- lpNetResource — указатель на структуру NETRESOURCE. Если этот параметр равен нулю, то перечисление начинается с самой верхней ступени иерархии сетевых ресурсов. Ноль ставится для того, чтобы получить самый первый ресурс. Потом я передаю в качестве этого параметра указатель на уже найденный ресурс. Тогда перечисление продолжится дальше. Так я повторяю, пока не найдутся все ресурсы;
- lphEnum — указатель, который понадобится в функции WnetEnumResource.

Теперь нужно рассмотреть структуру NETRESOURCE:

```
typedef struct _NETRESOURCE {
    DWORD  dwScope;
    DWORD  dwType;
    DWORD  dwDisplayType;
    DWORD  dwUsage;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    LPTSTR lpComment;
    LPTSTR lpProvider;
} NETRESOURCE;
```

Что такое dwScope, dwType и dwUsage, вы уже знаете. А вот остальные рассмотрим подробнее:

- dwDisplayType — способ отображения ресурса:
  - RESOURCEDISPLAYTYPE_DOMAIN — это домен;
  - RESOURCEDISPLAYTYPE_GENERIC — общий ресурс;
  - RESOURCEDISPLAYTYPE_SERVER — сервер;
  - RESOURCEDISPLAYTYPE_SHARE — разделяемый ресурс;

- `lpLocalName` — локальное имя;
- `lpRemoteName` — удаленное имя;
- `lpComment` — комментарий;
- `lpProvider` — хозяин ресурса. Параметр может быть равен нулю, если хозяин неизвестен.

Теперь можно переходить к следующей функции:

```
DWORD WNetEnumResource(
    HANDLE hEnum,           // Указатель на дескриптор перечисления
    LPDWORD lpcCount,       // Количество записей в буфере
    LPVOID lpBuffer,        // Буфер, куда будет записан результат
    LPDWORD lpBufferSize    // Размер буфера
   );
```

Параметры функции `WNetEnumResource`:

- `hEnum` — указатель на возвращенное функцией `WNetOpenEnum` значение;
- `lpcCount` — максимальное количество возвращаемых значений. Не стесняйтесь, ставьте 2000. Если вы зададите 0xFFFFFFFF, то перечислятся все ресурсы. После выполнения функция передаст сюда фактическое число найденных ресурсов;
- `lpBuffer` — указатель на буфер, в который будет помещен результат;
- `lpBufferSize` — размер буфера.

После окончания перечисления вызывается функция `WNetCloseEnum`, которая закрывает начатое функцией `WNetOpenEnum` перечисление сетевых ресурсов. В качестве единственного параметра нужно передать указатель на возвращенное функцией `WNetOpenEnum` значение.

Это все, что касается поиска открытых сетевых ресурсов. Осталось только сделать одно замечание. Функция поиска `WNetOpenEnum` и соответствующие ей структуры находятся в библиотеке mpr.lib, которая по умолчанию не линкуется к проекту. Чтобы собрать проект без ошибок, необходимо подключить эту библиотеку. Для этого щелкните правой кнопкой мыши по имени проекта в окне **Solution Explorer** и в появившемся меню выберите пункт **Properties**. Перед вами откроется окно свойств, в котором надо перейти в раздел **Configuration Properties/Linker/Input**. Здесь в строке **Additional Dependencies** напишите имя библиотеки mpr.lib (рис. 4.3).

Примечание

Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\NetNeighbour.

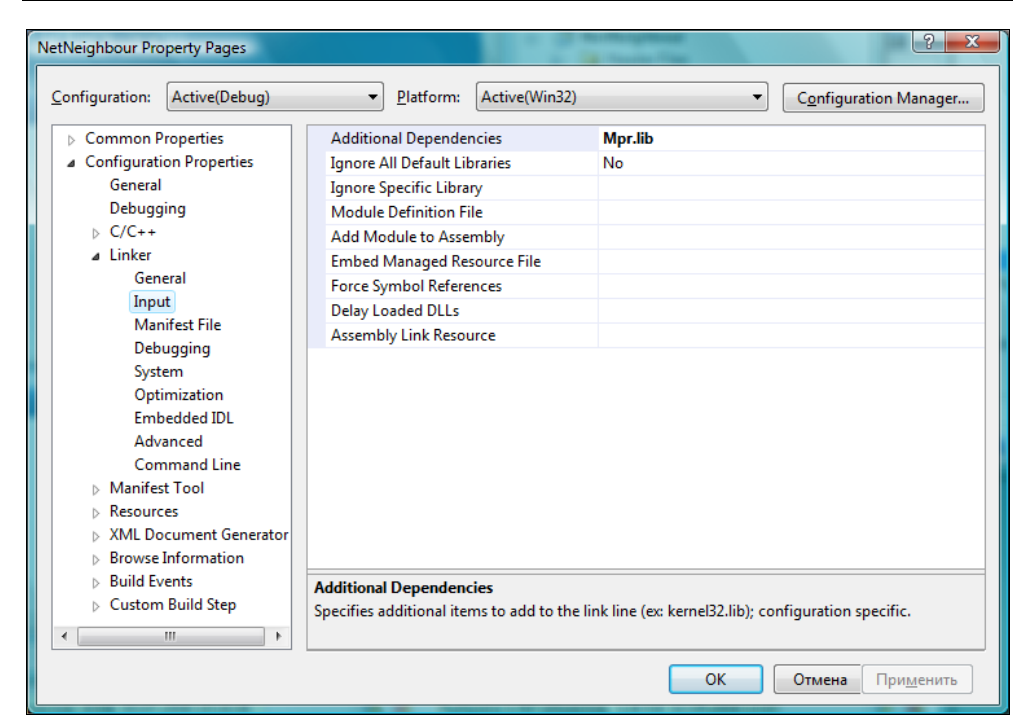


Рис. 4.3. Добавление библиотеки, содержащей функцию `WNetOpenEnum`

4.4. Работа с сетью с помощью объектов Visual C++

При работе с сетью можно использовать возможности, которые предоставляет среда разработки Visual C++. Объекты упрощают программирование и скрывают некоторые особенности реализации протоколов и сети.

При использовании объектов проекты будут достаточно большими, потому что уже нельзя использовать приложения Win32 Project. Проекты надо создавать с помощью Мастера MFC Application. Для начала этого будет достаточно, потому что основная цель сейчас — понять процесс программирования сетевых приложений. Чуть позже я познакомлю вас с сетевыми WinAPI-функциями, и тогда мы сможем написать те же приложения, но без использования объектов, и получить приложения маленького размера.

Для работы с сетью в MFC есть очень удобный класс — `CSocket`. В качестве предка у него выступает `CAsyncSocket`. Что это означает? Объект `CAsyncSocket` работает с сетью асинхронно. Отправив пакет в сеть, объект не ждет подтверждения, не блокирует выполнение, а программа может продолжать рабо-

тать дальше. Об окончании действия мы можем узнать по событиям, которые для нас уже реализованы в объекте, и достаточно только написать их обработчики.

При синхронной работе каждая отправка пакета или соединение с сервером замораживает выполнение программы до окончания выполнения действия. Таким образом, процессорное время расходуется нерационально. Нет, лишней нагрузки на процессор это не дает, но то, что в это время мы могли загрузить процессор полезными действиями — это факт.

Объект `CSocket` является потомком объекта `CAsyncSocket`, а значит, получает все его возможности, свойства и методы. Его работа построена на основе технологии "клиент-сервер". В принципе, это ограничение самого протокола TCP/IP. Это значит, что один объект может быть сервером, который принимает соединения клиентов и работает с ними. Из этого следует, что в примерах для передачи данных понадобится создавать два объекта: `CServerSocet` (сервер) и `CClientSocket` (клиент для подключения к серверу).

Объект `CServerSocket` схож с `CClientSocket`. Сервер ожидает соединения определенным портом, и когда клиент подключился, создается объект `CClientSocket`, с помощью которого можно отправлять и принимать данные на сервере.

Чтобы увидеть на практике работу с сетью, давайте напишем программу, которая будет сканировать указанный компьютер и искать на нем открытые порты (сканер портов). Как это работает? Для того чтобы узнать, какие порты открыты, достаточно только попробовать подсоединиться к порту. Если соединение пройдет успешно, значит, данный порт открыла какая-то программа.

Некоторые считают, что если какая-то серверная программа требует авторизации при соединении, то нельзя будет и присоединиться к ее порту. Это ошибочное мнение, потому что авторизация происходит только после соединения с сервером. Именно поэтому есть возможность определить все открытые порты, но только если между нами и сканируемым компьютером не установлен какой-нибудь специальный защитный комплекс (Firewall).

Теперь перейдем к делу. Создайте новый проект MFC Application в Visual C++ и назовите его MFCScan. В мастере измените следующие параметры:

- в разделе **Application Type** — установите тип приложения **Dialog based** (рис. 4.4);
- в разделе **Advanced Features** — выделите пункт **Windows Sockets**.

Теперь можно нажимать кнопку **Finish**, чтобы Visual C++ сформировал необходимые файлы.

Уже готов шаблон приложения, но еще не хватает объектов, через которые приложение будет работать с сетью. В принципе можно обойтись и без них,

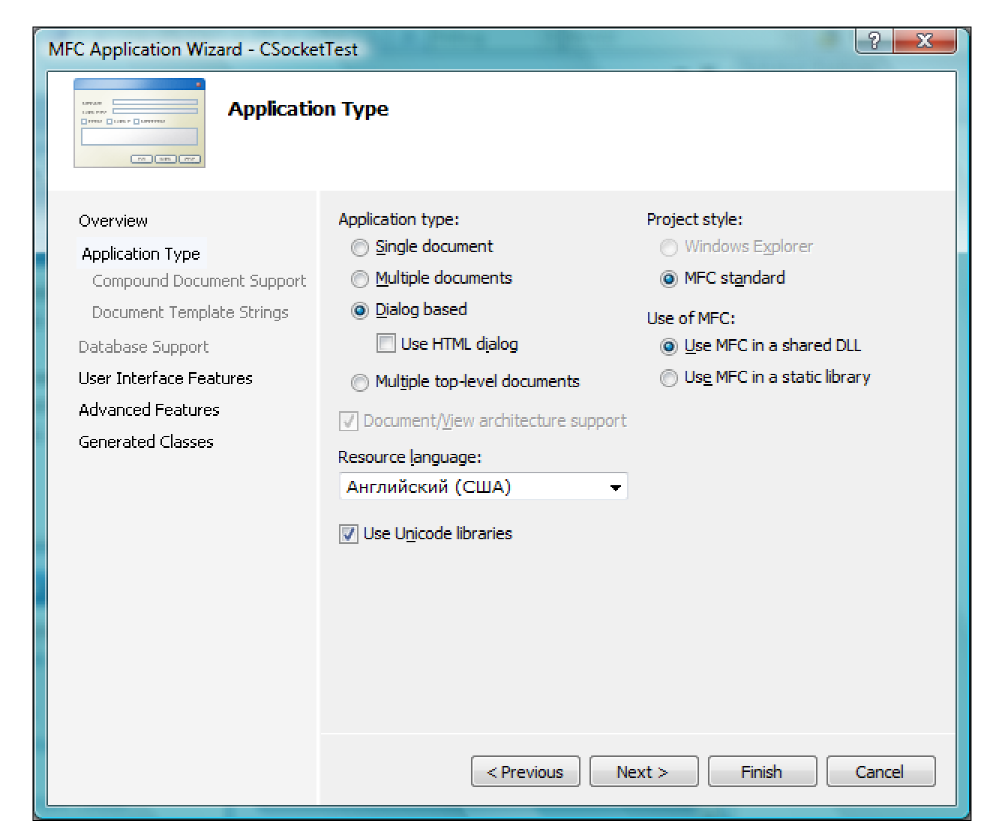


Рис. 4.4. Окно выбора типа приложения

но лучше добавить на тот случай, если вы захотите в будущем расширить возможности сканера. Щелкните правой кнопкой на имени проекта в окне **Solution Explorer** и выберите в появившемся меню **Add/Add Class...**. Перед вами откроется окно добавления класса (рис. 4.5). Выберите пункт **MFC Class** и нажмите кнопку **Open**.

Если вы все сделали правильно, то должно открыться окно, как на рис. 4.6. Здесь вы должны указать будущее имя класса и базовый класс, от которого будет происходить создаваемый. Для этого в списке **Base class** (Базовый класс) найдите имя **CSocket**, а в поле **Class name** (Имя создаваемого класса) введите `CClientSocket`.

Теперь в проект добавилось два новых файла: ClientSocket.cpp и ClientSocket.h. С их помощью можно управлять соединением, но пока только посмотрите на них.

Откройте файл ClientSocket.h и перейдите в начало описания класса. В окне **Properties** нажмите кнопку **Overrides**. В окне свойств появятся методы и

Рис. 4.5. Окно добавления класса в приложение

Рис. 4.6. Окно настроек класса

события (рис. 4.7), которые можно переписать, чтобы объект работал так, как необходимо вам. Для этого щелкните по выпадающему списку напротив нужного метода или события и выберите пункт меню **<Add> Имя метода**. Для данного метода я не буду ничего изменять.

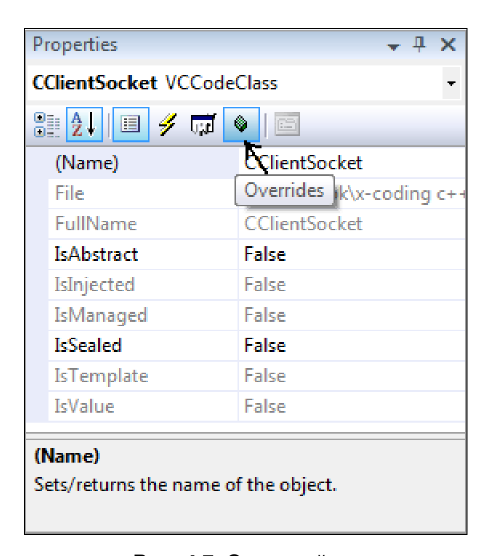


Рис. 4.7. Окно свойств

Теперь откройте файл ресурсов и найдите диалоговое окно `IDD_MFCSCAN_DIALOG`. Дважды щелкните по нему, чтобы откорректировать в редакторе ресурсов. Удалите кнопки **OK** и **Cancel**, поместите на окно диалога следующие компоненты:

- `Static Text` — с надписью "Server address";
- `Edit Control` — для ввода адреса сканируемого сервера (по умолчанию текст "Sample edit box");
- `List Box` — для сохранения открытых портов;
- `Button` (кнопка) — с надписью "Scan" для запуска сканирования портов указанного компьютера.

У вас должно получиться нечто похожее на рис. 4.8.

Теперь необходимо создать переменную для списка, чтобы с ним потом работать. Для этого надо щелкнуть по компоненту **List Box** правой кнопкой мыши и в выпадающем меню выбрать пункт **Add Variable**. В появившемся окне (рис. 4.9) нужно ввести в поле **Variable name** имя переменной. Укажите имя `PortsList`.

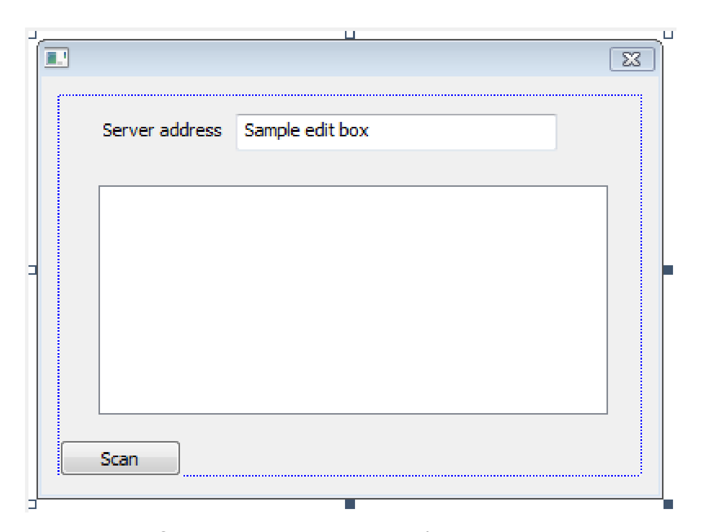


Рис. 4.8. Окно диалога для нашего будущего приложения

Add Member Variable Wizard - MFCScan

Welcome to the Add Member Variable Wizard

Access: public

Control variable

Variable type: CListBox

Control ID: IDC_LIST1

Category: Control

Variable name: PortsList

Control type: LISTBOX

Max chars:

Min value:

Max value:

.h file:

.cpp file:

Comment (// notation not required):

Finish

Cancel

Рис. 4.9. Окно создания переменной

Все подготовительные работы закончены. Можно приступать к написанию кода сканера портов. Необходимо создать обработчик события, который будет срабатывать при нажатии пользователем кнопки **Scan**, и написать в нем весь необходимый код. Для этого щелкните правой кнопкой мыши по компоненту **Button** и выберите в появившемся меню пункт **Add Event Handler**. Перед вами откроется окно мастера **Event Handler Wizard** (рис. 4.10). Согласитесь со всеми установками мастера и нажмите кнопку **Add and Edit**.

Мастер создаст заготовку в виде пустой функции для обработчика события. В ней нужно написать код из листинга 4.5.

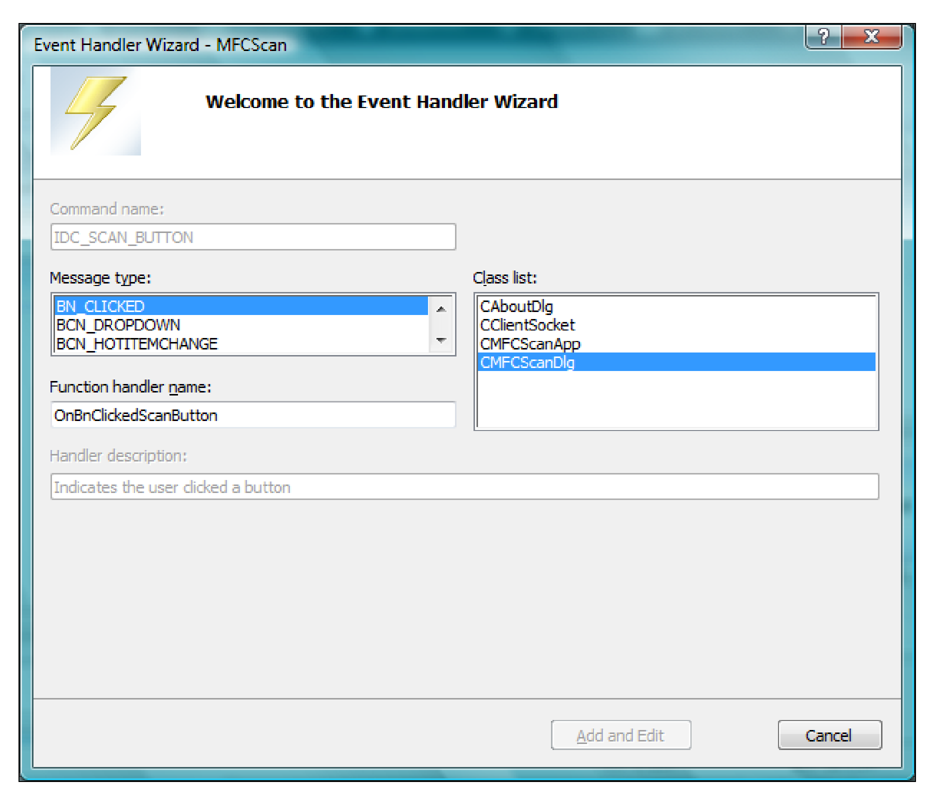


Рис. 4.10. Окно Мастера создания обработчика события

Листинг 4.5. Код сканера портов

```
void CMFCScanDlg::OnBnClickedScanButton()
{
	// TODO: Add your control notification handler code here
	CClientSocket *pSocket;
```

```
        CString       ip;
        CString messtr;
        int port;

        pSocket=new CClientSocket();
        pSocket->Create();

        GetDlgItemText(IDC_EDIT1,ip);
        port=1;
        while (port<100)
        {
               if(pSocket->Connect(ip, port))
               {
                      messtr.Format("Port=%d  opened", port);
                      PortsList.AddString(messtr);
                      pSocket->Close();
                      pSocket->Create();
               }
               port++;
        }
}
```

Теперь разберемся с тем, что здесь происходит. В данном коде объявлена переменная `pSocket` типа `CClientSocket`. С ее помощью мы будем работать с объектом, который умеет общаться с сетью по протоколу TCP/IP. Но прежде чем начать работу, нужно выделить память и создать объект. Это делается в следующих двух строчках:

```
pSocket = new CClientSocket();
pSocket->Create();
```

После этого следует узнать, какой IP-адрес указал пользователь в поле ввода. Для этого используется функция `GetDlgItemText`, у которой два параметра: идентификатор компонента и переменная, в которой будет сохранен результат.

Можно получить данные и с помощью специальной переменной. Для этого нужно было бы щелкнуть по компоненту в редакторе ресурсов правой кнопкой мыши и создать переменную. Но т. к. мы получаем данные только один раз, заводить переменную не имеет смысла.

После этого в переменную `port` заносится начальное значение 1, с которого начинается сканирование. Затем запускается цикл, который будет выполняться, пока переменная `port` не станет больше 100.

Внутри цикла производится попытка соединения с сервером следующим образом:

```
pSocket->Connect(ip, port)
```

Здесь вызывается метод `Connect` объекта, на который указывает переменная `pSocket`. У этого метода два параметра: адрес компьютера, к которому надо подключиться, и порт. Если соединение прошло удачно, то результатом будет ненулевое значение. В этом случае надо добавить информационную строку в список `PortsList`. Очень важно закрыть соединение и проинициализировать объект заново, иначе дальнейшие попытки соединения с сервером будут бесполезны, и вы увидите только первый открытый порт. Закрытие соединения и инициализация производятся методами `Close` и `Create` соответственно:

```
pSocket->Close();
pSocket->Create();
```

В конце цикла увеличивается переменная `port`, чтобы на следующем этапе цикла сканировать новый порт.

Теперь вы готовы скомпилировать программу, но чтобы все прошло удачно, нужно перейти в начало модуля, где перечислены подключаемые заголовочные файлы, и добавить следующую строку:

```
#include "ClientSocket.h"
```

Был использован объект `CClientSocket`, который описан в файле ClientSocket.h, поэтому без подключения модуля код не скомпилируется.

Запустите программу и, указав в качестве адреса 127.0.0.1, просканируйте порты своего компьютера, начиная с 0 до 99. Почему сканируем так мало портов? В Windows процесс сканирования 1000 портов происходит слишком медленно (может занять около 5 минут), поэтому сканировать лучше маленькими порциями.

Чуть позже я покажу более совершенный пример сканирования портов, а данная программа является чисто познавательной и очень хорошо подходит для понимания алгоритма сканирования. Если у вас большой опыт программирования в среде Visual C++ и вы знакомы с потоками, то я все равно не советую вам создавать множество потоков, чтобы каждый из них сканировал свой порт. Таким способом вы ускорите программу, но во время сканирования система будет нагружена, и причем бесполезно. Потерпите немного, и вы познакомитесь с реально быстрым сканером портов.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\Scan.

4.5. Передача данных по сети с помощью *CSocket*

Как я уже говорил, работа с сокетами происходит по технологии "клиент-сервер". Сервер запускается по определенному порту и начинает ожидать соединение. Клиент подключается на этот порт, и после этого может обмениваться данными с сервером.

Посмотрим, как передача данных выглядит на практике. Создайте новый проект MFC Application и назовите его `MFCSendText`. В Мастере измените параметры так же, как и в предыдущем примере со сканером портов *(см. разд. 4.4)*. Точно так же добавьте класс из `TSocket`. Точнее сказать, два класса: один для клиента, а другой — для сервера, и будут они называться `CClientSocket` и `CServerSocket` соответственно. Как видите, из одного класса `CSocket` выводятся оба класса: для сервера и для клиента.

Теперь оформим главное окно программы. Для этого откройте в редакторе ресурсов диалоговое окно **IDD_MFCSENDTEXT_DIALOG** и поместите на него четыре кнопки с заголовками **Create Server** (`IDC_BUTTON1`), **Connect to Server** (`IDC_BUTTON2`), **Send Data** (`IDC_BUTTON3`), **Disconnect** (`IDC_BUTTON4`). Внизу окна поместите `Static Text` для вывода сообщений.

Для кнопки **Send Data** создайте переменную. Для этого надо щелкнуть по ней правой кнопкой мыши и в появившемся меню выбрать пункт **Add Variable**. В окне Мастера создания переменной в поле **Variable name** укажите `m_SendButton`.

Теперь переходим к программированию. Для начала рассмотрим файл ServerSocket.h, в котором находится объявление класса `CServerSocket`. Его содержимое вы можете увидеть в листинге 4.6.

Листинг 4.6. Содержимое файла ServerSocket.h

```
#pragma once

#include "MFCSendTextDlg.h"

// CServerSocket command target
// (Определение класса CServerSocket)

class CServerSocket : public CSocket
{
public:
        CServerSocket(CMFCSendTextDlg* Dlg);
```

```
        virtual ~CServerSocket();
        virtual void OnAccept(int nErrorCode);
protected:
        CMFCSendTextDlg* m_Dlg;
};
```

Первое, что я изменил, — это конструктор. Теперь `CServerSocket` имеет один параметр `Dlg` типа `CMFCSendTextDlg`. Через этот параметр будет передаваться указатель на основной класс, чтобы была возможность обращаться к нему из класса `CServerSocket`. В разделе `protected` объявлена переменная для хранения указателя на класс главного окна.

Я также добавил в класс метод `OnAccept`, который будет вызываться, когда произойдет подключение к серверу.

В листинге 4.7 вы можете увидеть содержимое файла ServerSocket.cpp. Это реализация класса `CServerSocket`.

Листинг 4.7. Реализация класса `CServerSocket`

```
// ServerSocket.cpp : implementation file
// (Исполняемый файл)

#include "stdafx.h"
#include "MFCSendText.h"
#include "ServerSocket.h"

CServerSocket::CServerSocket(CMFCSendTextDlg* Dlg)
{
   m_Dlg = Dlg;
}

CServerSocket::~CServerSocket()
{
}

// CServerSocket member functions
// (Функции-члены объекта CServerSocket)
void CServerSocket::OnAccept(int nErrorCode)
{
   // TODO: Add your specialized code here and/or call the base class
   // (Добавьте сюда ваш код или/и вызовите базовый класс)
   AfxMessageBox("New connection accepted");
   m_Dlg->AddConnection();

   CSocket::OnAccept(nErrorCode);
}
```

В конструкторе `CServerSocket` просто сохраняется значение, полученное через единственный параметр `Dlg` в переменной `m_Dlg`.

Метод `OnAccept` вызывается каждый раз, когда клиент подключается к серверу. Здесь сначала с помощью функции `AfxMessageBox` выводится на экран сообщение о том, что принято новое соединение. Потом вызывается метод `AddConnection` класса окна, на который указывает переменная `m_Dlg`.

В данном случае я уверен, что переменная `m_Dlg` содержит действительные данные и указывает на существующий класс. Если вы считаете, что переменная может изменить значение или класс, на который она указывает, может быть уничтожен раньше времени, то желательно ввести проверку на корректность содержимого `m_Dlg` до вызова метода `AddConnection`. Это очень важно, иначе программа будет выполнять недопустимые операции и может нарушить работу системы.

Теперь рассмотрим содержимое файла ClientSocket.h, в котором находится описание класса `CClientSocket`, отвечающего за клиентскую часть (листинг 4.8).

Листинг 4.8. Содержимое файла ClientSocket.h

```
#pragma once

#include "MFCSendTextDlg.h"

// CClientSocket command target
// (Определение класса CClientSocket)

class CClientSocket : public CSocket
{
public:
	CClientSocket(CMFCSendTextDlg* Dlg);
	virtual ~CClientSocket();
	virtual void OnReceive(int nErrorCode);
	virtual void OnClose(int nErrorCode);
protected:
	CMFCSendTextDlg* m_Dlg;
};
```

Здесь также модифицирован конструктор, чтобы сохранять информацию о классе, создавшем класс клиента `CClientSocket`. Для этого заведена такая же переменная `m_Dlg`.

Помимо этого введены два метода: `OnReceive` (вызывается, когда по сети пришли новые данные) и `OnClose` (вызывается, когда соединение завершено).

Теперь посмотрим, как все это реализовано в файле ClientSocket.cpp (листинг 4.9).

Листинг 4.9. Содержимое файла ClientSocket.cpp

```
// ClientSocket.cpp : implementation file

#include "stdafx.h"
#include "MFCSendText.h"
#include "ClientSocket.h"


// CClientSocket

CClientSocket::CClientSocket(CMFCSendTextDlg* Dlg)
{
      m_Dlg = Dlg;
}

CClientSocket::~CClientSocket()
{
}

void CClientSocket::OnReceive(int nErrorCode)
{
    char recstr[1000];
    int r=Receive(recstr,1000);
    recstr[r]='\0';
    m_Dlg->SetDlgItemText(IDC_STATIC, recstr);

    CSocket::OnReceive(nErrorCode);
}

void CClientSocket::OnClose(int nErrorCode)
{
    m_Dlg->m_SendButton.EnableWindow(FALSE);
    CSocket::OnClose(nErrorCode);
}
```

Самое важное находится в методе `OnReceive`. Он вызывается каждый раз, когда для клиента пришли по сети какие-то данные. Для чтения полученных данных используется метод `Receive`. У него два параметра:

- буфер, в который будут записаны полученные данные, — переменная `recstr`;
- размер буфера.

Метод возвращает количество полученных по сети данных. Это значение записывается в переменную `r`. Теперь в переменной `recstr` находятся полученные данные, но по правилам языка C строки должны заканчиваться нулевым символом. Добавим его в буфер за последним полученным символом:

```
recstr[r]='\0';
```

Теперь полученный текст копируем в компонент `Static Text` на диалоговом окне с помощью следующей строки кода:

```
m_Dlg->SetDlgItemText(IDC_STATIC, recstr);
```

Метод `OnClose` вызывается каждый раз, когда соединение завершено. В его коде кнопку **Send Data** надо сделать недоступной, потому что без соединения с сервером нельзя отправлять данные.

```
m_Dlg->m_SendButton.EnableWindow(FALSE);
```

Сейчас перейдем к рассмотрению главного модуля программы — `MFCSendTextDlg`. Начнем разбор с заголовочного файла (листинг 4.10).

Листинг 4.10. Заголовочный файл MFCSendTextDlg.h

```
// MFCSendTextDlg.h : header file

#pragma once
#include "afxwin.h"

class CServerSocket;
class CClientSocket;

class CMFCSendTextDlg : public CDialog
{
// Construction  (Конструктор)
public:
	CMFCSendTextDlg(CWnd* pParent = NULL); // standard constructor
	                                       // (стандартный конструктор)

	// Dialog Data  (Данные диалога)
	enum { IDD = IDD_MFCSENDTEXT_DIALOG };

protected:
	// DDX/DDV support (Поддержка обмена данными)
	virtual void DoDataExchange(CDataExchange* pDX);


protected:
	HICON m_hIcon;
	CServerSocket* m_sSocket;
```

```
        CClientSocket* m_cSocket;
        CClientSocket* m_scSocket;

        // Generated message map functions
        // (Сгенерированные функции карты сообщений)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
        afx_msg HCURSOR OnQueryDragIcon();
        DECLARE_MESSAGE_MAP()
      public:
        afx_msg void OnBnClickedButton1();
        afx_msg void OnBnClickedButton2();
        afx_msg void OnBnClickedButton3();
        CButton m_SendButton;
        afx_msg void OnBnClickedButton4();
        void AddConnection();
};
```

Здесь введены три переменные в разделе `protected`:

- `m_sSocket` — указатель на класс `CServerSocket`;
- `m_cSocket` и `m_scSocket` — указатели на класс `CClientSocket`.

А в разделе `public` добавлен один метод `void AddConnection()`.

Теперь создайте поочередно обработчики события для всех кнопок диалогового окна. Для этого необходимо щелкнуть на кнопке правой кнопкой мыши и в появившемся меню выбрать пункт **Add Event Handler**. Давайте рассмотрим каждый обработчик события в отдельности.

Для кнопки **Create Server** будет следующий обработчик:

```
void CMFCSendTextDlg::OnBnClickedButton1()
{
    // TODO: Add your control notification handler code here
    m_sSocket=new CServerSocket(this);
    m_sSocket->Create(22345);
    m_sSocket->Listen();
    SetDlgItemText(IDC_STATIC, "Server started");
}
```

Здесь необходимо создать сервер и запустить прослушивание порта (ожидание соединений клиентов). В первой строке инициализируется переменная `m_sSocket`. Она имеет тип класса `CServerSocket`, поэтому в качестве параметра надо передать конструктору указатель на текущий класс. Это делается с помощью ключевого слова `this`.

После этого вызывается метод `Create`, у которого в качестве единственного параметра необходимо указать номер порта, на котором будет работать сервер. Теперь можно запускать прослушивание с помощью метода `Listen`.

Сервер запущен и ожидает подключения, и через компонент `Static Text` в окне диалога выводится соответствующее сообщение.

В обработчике события для кнопки **Connect To Server** надо написать следующий код:

```
void CMFCSendTextDlg::OnBnClickedButton2()
{
    // TODO: Add your control notification handler code here
    m_cSocket = new CClientSocket(this);
    m_cSocket->Create();
    if (m_cSocket->Connect("127.0.0.1", 22345))
          m_SendButton.EnableWindow(TRUE);
}
```

В первой строке инициализируется переменная `m_cSocket`. Следующей строкой кода создается класс. Теперь можно соединяться с сервером. Для этого используется метод `Connect`. Существует несколько реализаций данного метода, и они отличаются количеством и типом передаваемых параметров. В нашем случае используются следующие параметры:

- IP-адрес в виде строки;
- порт, на который необходимо подключиться.

Если соединение прошло успешно, то метод вернет ненулевое значение. Проверяется результат, и если все нормально, то кнопка **Send Data** делается доступной.

Отправка данных происходит, когда пользователь нажимает кнопку **Send Data**. Код, который должен находиться в обработчике события, выглядит следующим образом:

```
void CMFCSendTextDlg::OnBnClickedButton3()
{
    // TODO: Add your control notification handler code here
    m_cSocket->Send("Hello", 100);

    int err=m_cSocket->GetLastError();
    if(err>0)
    {
        CString ErrStr;
        ErrStr.Format("errcode=%d",err);
```

```
        AfxMessageBox(ErrStr);
    }
}
```

Отправка данных происходит с помощью метода `Send` объекта-клиента `m_cSocket`. У него два параметра:

- данные, которые надо отправить, — строка "Hello";
- размер данных — в данном случае нужно было бы указать 5, потому что отправляемое слово содержит 5 символов (5 байтов), но в примере указано 100. Это не приведет к ошибке, но позволит вам легко изменять отправляемую строку. В реальных приложениях обязательно указывайте истинную длину строки.

До этого я практически не проверял производимые действия на ошибки. А при запуске сервера это делать необходимо, потому что если сервер уже запущен, то повторная попытка приведет к ошибке. Кроме того, может возникнуть ситуация, когда на компьютере пользователя не установлен протокол TCP, тогда тоже будут ошибки. Правда, такая ситуация уже представляется с трудом, ибо сейчас, кажется, любой компьютер содержит соединение или установленный протокол.

При отправке данных такая проверка есть. С помощью метода `GetLastError` можно получить код ошибки последней операции в классе `m_cSocket`. Если результат метода `GetLastError` больше нуля, то была ошибка.

В обработчике события кнопки **Disconnect** выполняется следующий код:

```
void CMFCSendTextDlg::OnBnClickedButton4()
{
    // TODO: Add your control notification handler code here
    SetDlgItemText(IDC_STATIC, "Disconnected");
    m_cSocket->Close();
}
```

Первой строкой в текстовое поле в окне диалога выводится сообщение о том, что соединение разорвано. Во второй строке вызывается метод `Close`, который закрывает соединение с сервером.

Теперь самое интересное — метод `AddConnection`, который я уже использовал, когда произошло соединение с сервером. Посмотрим, что в нем происходит:

```
void CMFCSendTextDlg::AddConnection()
{
    m_scSocket = new CClientSocket(this);
    m_sSocket->Accept(*m_scSocket);
}
```

Как видите, здесь создается новый объект типа CClientSocket. После этого он присоединяется к серверу m_sSocket методом Accept. Так переменная класса CclientSocket связывается с новым соединением. Именно через эту переменную сервер может отослать данные к клиенту, и именно через нее он принимает данные.

Получается, что один класс CClientSocket используется на клиенте для соединения с сервером и отправки ему данных, а на сервере — для получения и возврата данных. Класс CServerSocket используется только для прослушивания порта и получения соединения.

Данный пример будет хорошо работать только тогда, когда один клиент соединяется с сервером. Если второй клиент попытается соединиться, то переменная m_scSocket будет перезаписана для нового клиента. Именно поэтому на сервере вы должны хранить динамический массив классов типа CClientSocket. При подключении клиента вы должны создавать новый класс типа CClientSocket и сохранять его в массиве, а при отключении клиента соответствующий класс должен уничтожаться и удаляться из массива.

Напоследок хочется заметить, что я нигде не указывал протокол, по которому будут работать клиент с сервером. По умолчанию класс CSocket использует TCP/IP.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\MFCSendText.

4.6. Работа напрямую с WinSock

Как видите, работа с сетью с использованием MFC-объектов, а именно CSocket, очень проста. Но вы не сможете таким образом написать маленькое приложение, потому что для этого надо отказаться от использования стандартных библиотек. Именно поэтому я рассмотрю работу с сетью напрямую достаточно подробно.

В Windows для работы с сетью используется библиотека WinSock. Существуют две версии этой библиотеки. Первая версия WinSock разрабатывалась на основе модели сокетов Беркли, используемой в UNIX-системах. Начиная с Windows 98, в ОС уже встроена вторая версия, улучшенная компанией Microsoft и расширенная множеством интересных функций.

Библиотека WinSock обратно совместима. Это значит, что старые функции не изменились, и программы, написанные для первой версии, будут прекрасно работать во второй. В более поздних версиях добавились новые функции, но

они оказались несовместимы с сетевыми функциями на других платформах. Впервые новшества появились в версии 1.1, и это были `WSAStartup`, `WSACleanup`, `WSAGetLastError`, `WSARecvEx` (имена начинаются с "WSA"). В следующей версии таких функций стало намного больше.

Если вам доступна версия WinSock2, то не обязательно ее использовать. Посмотрите, может быть возможностей первой версии окажется достаточно, и тогда вашу программу будет легко адаптировать к компилированию на платформе UNIX.

Конечно, компьютеры с установленной Windows 95 встретить уже достаточно сложно, но они существуют. Если вы обладатель такой ОС, то вы можете скачать новую версию библиотеки с сайта **www.microsoft.com**. Если вы решили использовать в своей программе первую версию, то необходимо подключить заголовочный файл winsock.h, иначе — winsock2.h.

Если проблема переносимости кода на другую платформу для вас не стоит, то я рекомендую не обращать внимания на единичные компьютеры с Windows 95, которые где-то еще могут встретиться. Смело выбирайте использование Windows Socket 2.0 и используйте сеть по максимуму.

4.6.1. Обработка ошибок

В самом начале необходимо узнать, как можно определить ошибки, которые возникают при вызове сетевых функций. Правильная обработка ошибок для любого приложения является самым важным. Хотя сетевые функции не критичны для ОС, но могут повлиять на ход работы программы. А это в свою очередь может привести к снижению безопасности системы.

Сетевые приложения обмениваются данными с чужими компьютерами, а это значит, что в качестве стороннего клиента может выступить злоумышленник. Если не обработать ошибку, это может привести к доступу к важным данным или функциям. Если не проверять буферы данных, то хакер может получить возможность сверхопасного удаленного выполнения кода в вашей системе.

Приведу простейший пример. У вас есть функция, которая вызывается каждый раз, когда программе пришли данные, и проверяет их на корректность и права доступа. Если все нормально, то функция выполняет критический код, который не должен быть доступен злоумышленнику. Контроль должен происходить на каждом этапе: получение данных, проверка их корректности и доступности, а также любой вызов сетевой функции. Помните, что это придаст стабильность и надежность вашей программе и обеспечит безопасность всей системы.

Если во время выполнения какой-либо функции произошла ошибка, то она вернет константу `SOCKET_ERROR` или `-1`. Если вы получили такое значение, то

можно воспользоваться функцией WSAGetLastError. Ей не надо передавать параметры, она просто возвратит код ошибки, которая произошла во время выполнения последней сетевой функции. Кодов ошибок очень много, и они зависят от функции, которая отработала последней. Я буду рассматривать их по мере необходимости.

4.6.2. Запуск библиотеки

Прежде чем начать работу с сетью, нужно загрузить необходимую версию библиотеки. В зависимости от этого изменяется набор доступных функций. Если использовать первую версию библиотеки, а вызвать функцию из второй, то произойдет сбой в работе программы. Если не загрузить библиотеку, то любой вызов сетевой функции вернет ошибку WSANOTINITIALISED.

Для загрузки библиотеки используется функция WSAStartup, которая выглядит следующим образом:

```
int WSAStartup (
    WORD wVersionRequested,
    LPWSADATA lpWSAData
   );
```

Первый параметр (wVersionRequested) — это запрашиваемая версия библиотеки. Младший байт указываемого числа определяет основной номер версии, а старший байт — дополнительный номер. Чтобы легче было работать с этим параметром, я советую использовать макрос MAKEWORD(i,j), где i — это старший байт, а j — младший.

Второй параметр функции WSAStartup — это указатель на структуру WSADATA, в которой после выполнения функции будет находиться информация о библиотеке.

Если загрузка прошла успешно, то результат будет нулевым, иначе произошла ошибка.

Посмотрите пример использования функции WSAStartup для загрузки библиотеки WinSock 2.0:

```
WSADATA wsaData;

int err = WSAStartup(MAKEWORD(2, 0), &wsaData);
if (err != 0)
{
    // Tell the user that WinSock not loaded
    // (Сказать пользователю, что библиотека не загружена)
    return;
}
```

Обратите внимание, что сразу после попытки загрузить библиотеку идет проверка возвращенного значения. Если функция отработала правильно, то она должна вернуть нулевое значение. Приведу основные коды ошибок:

- WSASYSNOTREADY — основная сетевая подсистема не готова к сетевому соединению;
- WSAVERNOTSUPPORTED — запрашиваемая версия библиотеки не поддерживается;
- WSAEPROCLIM — превышен предел поддерживаемых ОС задач;
- WSAEFAULT — неправильный указатель на структуру WSAData.

Структура WSADATA выглядит следующим образом:

```
typedef struct WSAData {
     WORD                wVersion;
     WORD                wHighVersion;
     char                szDescription[WSADESCRIPTION_LEN+1];
     char                szSystemStatus[WSASYS_STATUS_LEN+1];
     unsigned short      iMaxSockets;
     unsigned short      iMaxUdpDg;
     char FAR *          lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

Разберем каждый параметр в отдельности:

- wVersion — версия загруженной библиотеки WinSock;
- wHighVersion — последняя версия;
- szDescription — текстовое описание, которое заполняется не всеми версиями;
- szSystemStatus — текстовое описание состояния, которое заполняется не всеми версиями;
- iMaxSockets — максимальное количество открываемых соединений. Эта информация не соответствует действительности, потому что максимальное число зависит только от доступных ресурсов. Параметр остался только для совместимости с первоначальной спецификацией;
- iMaxUdpDg — максимальный размер дейтаграммы (пакета). Информация не соответствует действительности, потому что размер зависит от протокола;
- lpVendorInfo — информация о производителе.

Давайте рассмотрим небольшой пример, с помощью которого будет загружаться библиотека WinSock из структуры WSAData. Создайте новый проект MFC Application. В Мастере создания приложений, в разделе **Application**

Type выберите **Dialog based**, а в разделе **Advanced Features** — **Windows sockets**. Это уже знакомый вам тип приложения.

Откройте в редакторе ресурсов диалоговое окно и оформите, как на рис. 4.11. На форме должно быть 4 поля **Edit Control** для вывода информации о загруженной библиотеке и кнопка **Get WinSock Info**, по нажатию которой будут загружаться данные.

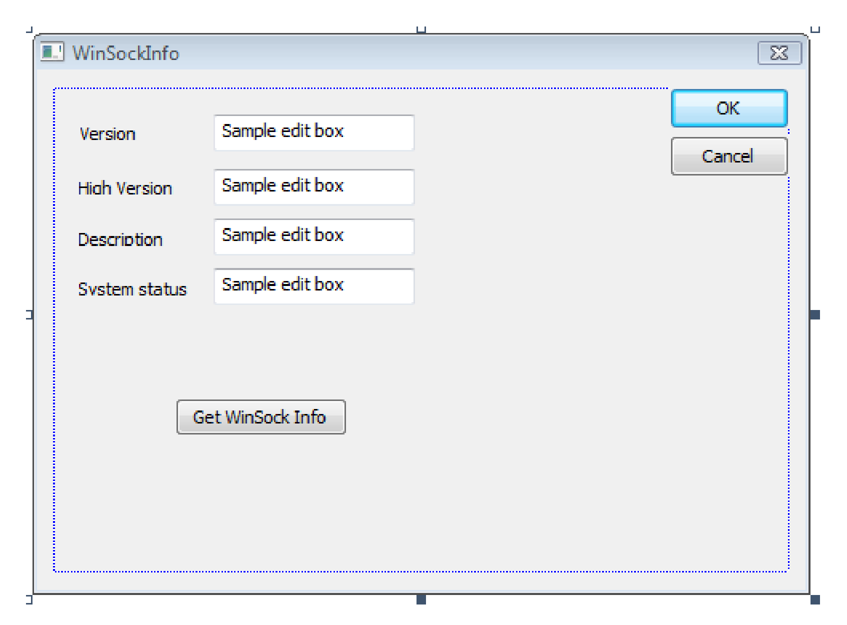


Рис. 4.11. Окно будущей программы WinSockInfo

Для каждого поля вводятся переменные:

- номер версии — `mVersion`;
- последняя версия — `mHighVersion`;
- описание — `mDescription`;
- состояние — `mSystemStatus`.

Создайте обработчик события для кнопки **Get WinSock Info** и напишите в нем код из листинга 4.11.

Листинг 4.11. Получение информации о WinSock

```
void CWinSockInfoDlg::OnBnClickedButton1()
{
   // TODO: Add your control notification handler code here
   WSADATA wsaData;
```

```
    int err = WSAStartup(MAKEWORD(2, 0), &wsaData);
    if (err != 0)
    {
        // Tell the user that WinSock not loaded
        return;
    }

    char mText[255];
    mVersion.SetWindowText(itoa(wsaData.wVersion, mText, 10));
    mHighVersion.SetWindowText(itoa(wsaData.wHighVersion, mText, 10));
    if (wsaData.szDescription)
         mDescription.SetWindowText(wsaData.szDescription);
    if (wsaData.szSystemStatus)
         mSystemStatus.SetWindowText(wsaData.szSystemStatus);
}
```

В самом начале запускается WinSock (код, который я уже приводил). После этого полученная информация просто выводится в поля на форме диалога.

На рис. 4.12 вы можете увидеть результат работы программы на моем компьютере.

Примечание

Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\WinSockInfo.

Рис. 4.12. Результат работы программы WinSockInfo

В приведенном примере есть один недостаток — не выгружается библиотека, потому что я поленился, и мы еще не рассматривали необходимую функцию, но пора исправить эту ситуацию. Для освобождения библиотеки используется функция `WSACleanup`:

```
int WSACleanup(void);
```

Функции не нужны параметры, она просто освобождает библиотеку, после чего работа с сетевыми функциями становится недоступной для данного приложения, поэтому ее вызывают по выходу из программы, когда работа с сетью уже не нужна.

4.6.3. Создание сокета

После загрузки библиотеки необходимо создать сокет, с помощью которого происходит работа с сетью. Для этого в первой версии библиотеки есть функция `socket`:

```
SOCKET socket (
    int af,
    int type,
    int protocol
   );
```

В версии WinSock2 для создания сокета можно использовать функцию `WSASocket`.

```
SOCKET WSASocket (
    int af,
    int type,
    int protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    GROUP g,
    DWORD dwFlags
   );
```

Первые три параметра и возвращаемое значение для обеих функций одинаковы. И в том, и в другом случае функция возвращает созданный сокет, который будет использоваться в дальнейшем при работе с сетью. Давайте рассмотрим общие параметры:

- `af` — семейство протоколов, которые можно использовать, например:
  - `AF_UNSPEC` — спецификация не указана;
  - `AF_INET` — интернет-протоколы TCP, UDP и т. д. В данной книге я буду использовать именно эти протоколы, как самые популярные и распространенные;

- `AF_IPX` — протоколы IPX, SPX;
- `AF_APPLETALK` — протокол AppleTalk;
- `AF_NETBIOS` — протокол NetBIOS;

- `type` — спецификация нового сокета. Здесь можно указывать одно из следующих значений:
  - `SOCK_STREAM` — передача с установкой соединения. Для интернет-протоколов будет использоваться TCP;
  - `SOCK_DGRAM` — передача данных без установки соединения. Для интернет-протоколов будет использоваться UDP;
- `protocol` — протокол для использования. Протоколов очень много, и вы можете узнать об используемых константах в справочной системе по программированию, а я чаще всего буду использовать константу `IPPROTO_TCP`, которая соответствует протоколу TCP.

В функции `WSASocket` добавлены еще три параметра:

- `lpProtocolInfo` — указатель на структуру `WSAPROTOCOL_INFO`, в которой определяются характеристики создаваемого сокета;
- `g` — идентификатор группы сокетов;
- `dwFlags` — атрибуты сокета.

Более подробно с указанными параметрами вы познакомитесь в процессе написания примеров. Это поможет вам лучше понять их и сразу же увидеть результат работы.

4.6.4. Серверные функции

Вы уже знаете, что протокол TCP работает по технологии "клиент-сервер". Чтобы два компьютера смогли установить соединение, один из них должен запустить прослушивание на определенном порту. И только после этого клиент может присоединиться к серверу.

Давайте рассмотрим функции, необходимые для создания сервера. Первым делом следует связать сетевой локальный адрес с уже созданным сокетом. Эта связь указывает, от какого из сетевых интерфейсов мы будем ожидать подключения, ведь с каждым сетевым интерфейсом связан адрес, и именно к нему можно привязаться. Мы можем указать, что подключения можно ожидать от любого сетевого интерфейса.

Для привязки используется функция `bind`:

```
int bind (
    SOCKET s,
```

```
    const struct sockaddr FAR*  name,
    int namelen
   );
```

Давайте посмотрим на параметры этой функции:

- предварительно созданный сокет;
- указатель на структуру типа `sockaddr`;
- размер структуры `sockaddr`, указанной в качестве второго параметра.

Структура `sockaddr` предназначена для хранения адреса, а в разных протоколах используется своя адресация. Поэтому и структура `sockaddr` может выглядеть по-разному. Для интернет-протоколов структура имеет имя `sockaddr_in` (мне кажется, in в конце имени символизирует слово internet) и выглядит следующим образом:

```
struct sockaddr_in {
    short   sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```

Рассмотрим параметры этой структуры:

- `sin_family` — семейство протоколов. Этот параметр схож с первым параметром функции `socket`. Для интернет-протоколов указывается константа `AF_INET`;
- `sin_port` — порт для идентификации программы поступающими данными;
- `sin_addr` — структура `SOCKADDR_IN`, которая хранит IP-адрес;
- `sin_zero` — используется для выравнивания адреса из параметра `sin_addr`. Это необходимо, чтобы размер структуры `SOCKADDR_IN` равнялся размеру `SOCKADDR`.

Сейчас я хочу подробнее остановиться на портах. Вы должны быть очень внимательны при выборе порта, потому что если он уже занят какой-либо программой, то вторая попытка открытия закончится ошибкой. Вы должны знать, что некоторые порты зарезервированы для определенных (наиболее популярных) служб. Номера этих портов распределяются центром Internet Assigned Numbers Authority. Существует три категории портов:

- 0–1023 — управляются IANA и зарезервированы для стандартных служб. Не рекомендуется использовать порты из этого диапазона, если только вы

не пишете собственную реализацию общепринятого сервиса, например, WEB, FTP или почтовый сервис;

- 1024–49151 — зарезервированы IANA, но могут использоваться процессами и программами. Большинство из этих портов можно использовать;
- 49152–65535 — частные порты, никем не зарезервированы.

Если во время выполнения функции `bind` выяснится, что порт уже используется какой-либо службой, то функция вернет ошибку `WSAEADDRINUSE`.

Давайте рассмотрим пример кода, который создает сокет и привязывает к нему сетевой локальный адрес:

```
SOCKET s=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(4888);
addr.sin_addr.s_addr=htonl(INADDR_ANY);

bind(s, (SOCKADDR*)&addr), sizeof(addr);
```

В данном примере создается сокет со следующими параметрами:

- `AF_INET` — означает, что будет использоваться семейство интернет-протоколов;
- `SOCK_STREAM` — указывает на протокол, устанавливающий соединение;
- `IPPROTO_TCP` — используется протокол TCP.

Затем объявляется структура `addr` типа `sockaddr_in`. В параметре `sin_family` структуры также указывается семейство интернет-протоколов (`AF_INET`). В параметре `sin_port` указывается номер порта. Байты в номере должны следовать в определенном порядке, который несовместим с порядком байтов в числовых переменных языка C и x86 процессорах, поэтому происходит преобразование с помощью функции `htons`.

В параметре `sin_addr.s_addr` указывается специальный адрес `INADDR_ANY`, который позволит в дальнейшем программе ожидать соединение на любом сетевом интерфейсе. Это значит, что если у вас две сетевые карты, соединенные с разными сетями, то программа будет ожидать соединения из обеих сетей.

После того как локальный адрес и порт привязаны к сокету, можно приступить к прослушиванию порта в ожидании соединения со стороны клиента. Для этого служит функция `listen`, которая выглядит следующим образом:

```
int listen (
    SOCKET s,
    int backlog
   );
```

Первый параметр — это все тот же сокет, который был создан и к которому привязан адрес. По этим данным функция определит, на каком порту нужно запустить прослушивание.

Второй параметр — это максимально допустимое число запросов, ожидающих обработки. Допустим, что вы указали здесь значение 3, а вам пришло 5 запросов на соединение от разных клиентов. Только первые три из них встанут в очередь, а остальные получат ошибку `WSAECONNREFUSED`, поэтому при написании клиента (в части соединения) обязательно должна быть проверка.

При вызове функции `listen` вы можете получить следующие основные ошибки:

- `WSAEINVAL` — функция `bind` не была вызвана для данного сокета;
- `WSANOTINITIALISED` — не загружена библиотека WinSock, т. е. не выполнена функция `WSAStartup`;
- `WSAENETDOWN` — нарушена сетевая подсистема;
- `WSAEISCONN` — сокет уже подключен.

Остальные ошибки возникают реже.

Когда клиент попадает в очередь на подключение к серверу, необходимо разрешить соединение с помощью функции `accept`. Она выглядит следующим образом:

```
SOCKET accept(
    SOCKET s,
    struct sockaddr FAR* addr,
    int FAR* addrlen
   );
```

Во второй версии есть функция `WSAAccept`, у которой первые три параметра такие же, как и у функции `accept`. Функция `WSAAccept` выглядит следующим образом:

```
SOCKET WSAAccept (
    SOCKET s,
    struct sockaddr FAR * addr,
    LPINT addrlen,
```

```
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData
   );
```

Давайте рассмотрим общие параметры для этих функций:

- предварительно созданный и запущенный на прослушивание сокет;
- указатель на структуру типа `sockaddr`;
- размер структуры `sockaddr`, указанной в качестве второго параметра.

После выполнения функции `accept` второй параметр (`addr`) будет содержать сведения об IP-адресе клиента, который произвел подключение. Эти данные можно использовать для контроля доступа к серверу по IP-адресу.

Функция `accept` возвращает указатель на новый сокет, который можно использовать для общения с клиентом. Старая переменная типа `SOCKET` продолжает слушать порт в ожидании новых соединений, и ее использовать нет смысла. Таким образом, для каждого подключенного клиента будет свой `SOCKET`, благодаря чему вы сможете работать с любым из них.

Если вы вспомните пример с передачей данных с использованием MFC-объектов *(см. разд. 4.5)*, то там применялся тот же метод. Как только клиент подключался к серверу, мы создавали новый сокет, через который и происходила работа с подключившимся клиентом. Именно этот сокет принимал данные, пришедшие по сети, и мог их отправлять обратно программе на стороне клиента. Серверный сокет используется только для прослушивания подключений и принятия их. На самом деле это очень удобно, в чем мы скоро убедимся.

4.6.5. Клиентские функции

В *разд. 4.6.4* я познакомил вас с серверными функциями, и вы уже в состоянии написать сервер с помощью WinAPI-функций. Но мы пока не будем этого делать, потому что еще не можем написать клиентскую часть и протестировать пример. Так что теперь самое время заняться клиентскими функциями и функциями приема/передачи данных, чтобы не просто создать сервер, но и протестировать собственной программой.

Для соединения с сервером нужны два этапа: создать сокет и подключиться к нему. Но это в идеальном случае. Как правило, добавляется еще один этап. Какой? Простым пользователям тяжело работать с IP-адресами, поэтому чаще всего они используют символьные имена серверов. В этом случае необходимо перед соединением с сервером выполнить еще одно действие — перевести символьное имя в IP-адрес. Имя может быть как именем компьютера, так и DNS.

Как создавать сокет, вы уже знаете. Теперь давайте разберемся с процессом определения IP-адреса. Для этого используется одна из двух функций: `gethostbyname` или `WSAAsyncGetHostByName` (в зависимости от версии WinSock). Для начала рассмотрим функцию `gethostbyname`:

```
struct hostent FAR * gethostbyname (
    const char FAR * name
   );
```

В качестве единственного параметра нужно передать символьное имя сервера. Функция возвращает структуру типа `hostent`, которую рассмотрим чуть позже.

Теперь переходим к рассмотрению `WSAAsyncGetHostByName`:

```
HANDLE WSAAsyncGetHostByName (
    HWND hWnd,
    unsigned int wMsg,
    const char FAR * name,
    char FAR * buf,
    int buflen
   );
```

Функция выполняется асинхронно, а это значит, что не блокируется выполнение программы при ее вызове. Программа будет работать дальше, но результат будет получен позже через сообщение Windows, указанное в качестве второго параметра. Это очень удобно, потому что процесс определения адреса может оказаться очень долгим, и блокирование программы на длительное время будет неэффективным. Процессорное время можно использовать для других целей.

Давайте разберем параметры подробнее:

- `hWnd` — дескриптор окна, которому будет послано сообщение по завершении выполнения асинхронного запроса;
- `wMsg` — сообщение Windows, которое будет сгенерировано после определения IP-адреса;
- `name` — символьное имя компьютера, адрес которого надо определить;
- `buf` — буфер, в который будет помещена структура `hostent`. Буфер должен иметь достаточный объем памяти. Максимальный размер можно определить с помощью макроса `MAXGETHOSTSTRUCT`;
- `buflen` — длина буфера, указанного в четвертом параметре.

Теперь рассмотрим структуру `hostent`, с помощью которой получен результат:

```
struct hostent {
    char FAR *        h_name;
    char FAR * FAR * h_aliases;
    short            h_addrtype;
    short            h_length;
    char FAR * FAR * h_addr_list;
};
```

Проанализируем параметры структуры:

- h_name — полное имя компьютера. Если в сети используется доменная система, то этот параметр будет содержать полное доменное имя;
- h_aliases — дополнительное имя узла;
- h_addrtype — тип возвращаемого адреса;
- h_length — длина каждого адреса в списке адресов;
- h_addr_list — список адресов компьютера.

Компьютер может иметь несколько адресов, поэтому структура возвращает полный список, который заканчивается нулем. В большинстве случаев достаточно выбрать первый адрес из списка. Если функция gethostbyname определила несколько адресов, то чаще всего по любому из них можно будет соединиться с искомым компьютером.

Теперь непосредственно функция соединения с сервером connect. Она выглядит следующим образом:

```
int connect (
    SOCKET s,
    const struct sockaddr FAR*  name,
    int namelen
   );
```

Параметры функции:

- s — предварительно созданный сокет;
- name — структура SOCKADDR, содержащая адрес сервера, к которому надо подключиться;
- namelen — размер структуры SOCKADDR, указанной в качестве второго параметра.

Во второй версии WinSock появилась функция WSAConnect:

```
int  WSAConnect (
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
```

```
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
   );
```

Первые три параметра ничем не отличаются от параметров функции `connect`. Поэтому рассмотрим только новые:

- ❒ `lpCallerData` — указатель на пользовательские данные, которые будут отправлены серверу во время установки соединения;
- ❒ `lpCalleeData` — указатель на буфер, в который будут помещены переданные во время соединения данные.

Оба параметра имеют тип указателя на структуру `WSABUF`, которая выглядит следующим образом:

```
typedef struct _WSABUF {
    u_long      len;
    char FAR *  buf;
} WSABUF, FAR * LPWSABUF;
```

Здесь первый параметр — размер буфера, а второй — указатель на сам буфер. Последние два параметра функции `WSAConnect` (`lpSQOS` и `lpGQOS`) являются указателями на структуры типа `QoS`. Они определяют требования к пропускной способности канала при приеме и передаче данных. Если указать нулевое значение, то это будет означать, что требования к качеству обслуживания не предъявляются.

Во время попытки соединения чаще всего могут встретиться следующие ошибки:

- ❒ `WSAETIMEDOUT` — сервер недоступен. Возможна какая-то проблема на пути соединения;
- ❒ `WSAECONNREFUSED` — на сервере не запущено прослушивание указанного порта;
- ❒ `WSAEADDRINUSE` — указанный адрес уже используется;
- ❒ `WSAEAFNOSUPPORT` — указанный адрес не может использоваться с данным сокетом. Эта ошибка возникает, когда указывается адрес в формате одного протокола, а производится попытка соединения по другому протоколу.

4.6.6. Обмен данными

Вы узнали, как создавать сервер, и познакомились с функциями соединения. Теперь необходимо научиться тому, ради чего все это задумывалось — пере-

давать и принимать данные. Именно ради обмена данными между компьютерами мы рассматривали такое количество функций.

Сразу замечу, что функции создавались тогда, когда еще не было даже разговоров о UNICODE (универсальная кодировка, позволяющая работать с любым языком). Поэтому, чтобы отправить данные в этой кодировке, нужно привести их к типу `char*`, а длину умножить на 2, потому что каждый символ в UNICODE занимает 2 байта (в отличие от ASCII, где символ равен одному байту). Тут важно отметить, что принимающая сторона должна знать, в какой кодировке вы отправляете данные, иначе все, кроме английских букв, может превратиться в абракадабру.

Чтобы принять данные, нужно их сначала отправить. Поэтому начну рассмотрение функций обмена данными с этого режима. Для передачи данных серверу существуют функции `send` и `WSASend` (для WinSock2). Функция `send` выглядит следующим образом:

```
int send (
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags
   );
```

Функция передает следующие параметры:

- `s` — сокет, через который будет происходить отправка данных. В программе может быть открыто одновременно несколько соединений с разными серверами или даже с одним сервером, и нужно четко определить, какой сокет надо использовать;
- `buf` — буфер, содержащий данные, которые необходимо отправить;
- `len` — длина буфера в параметре `buf`;
- `flags` — флаги, определяющие метод отправки. Здесь можно указывать сочетание из следующих значений:
  - `0` — флаги не указаны;
  - `MSG_DONTROUTE` — отправляемые пакеты не надо маршрутизировать. Если транспортный протокол, который отправляет данные, не поддерживает этот флаг, то он игнорируется;
  - `MSG_OOB` — данные должны быть отправлены вне очереди (out of band), т. е. срочно.

Функция `WSASend` выглядит следующим образом:

```
int WSASend (
    SOCKET s,
```

```
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
   );
```

Рассмотрим параметры этой функции:

- `s` — сокет, через который будет происходить отправка данных;
- `lpBuffers` — структура или массив структур типа `WSABUF`. С этой структурой вы познакомились, когда рассматривали функцию `connect`. Эта же структура использовалась для отправки данных во время соединения;
- `dwBufferCount` — количество структур в параметре `lpBuffers`;
- `lpNumberOfBytesSent` — количество переданных байтов для завершенной операции ввода/вывода;
- `dwFlags` — определяет метод отправки и может принимать такие же значения, как и параметр `dwFlags` функции `send`;
- `pOverlapped` и `pCompletionRoutine` — задаются при использовании перекрытого ввода/вывода (overlapped I/O). Это одна из моделей асинхронной работы сети, поддерживаемой WinSock.

Если функция `send` (или `WSASend`) отработала успешно, то она вернет количество отправленных байтов, иначе — значение –1 (или константу `SOCKET_ERROR`, которая равна –1). Получив ошибку, вы можете проанализировать ее с помощью функции `WSAGetLastError`:

- `WSAECONNABORTED` — соединение было разорвано, вышло время ожидания или произошла другая ошибка;
- `WSAECONNRESET` — удаленный компьютер разорвал соединение, и вам необходимо закрыть сокет;
- `WSAENOTCONN` — соединение не установлено;
- `WSAETIMEDOUT` — время ожидания ответа вышло.

Для получения данных используются функции `recv` и `WSARecv` (для второй версии WinSock). Функция `recv` выглядит следующим образом:

```
int recv (
    SOCKET s,
    char FAR* buf,
    int len,
    int flags
   );
```

Параметры очень похожи на те, которые описаны для функции `send`:

- `s` — сокет, данные которого надо получить;
- `buf` — буфер, в который будут помещены принятые данные;
- `len` — размер буфера в параметре `buf`;
- `flags` — флаги, определяющие метод получения. Здесь можно указывать сочетание из следующих значений:
  - `0` — флаги не указаны;
  - `MSG_PEEK` — считать данные из системного буфера без удаления. По умолчанию считанные данные стираются из системного буфера;
  - `MSG_OOB` — обработать срочные данные out of band.

Использовать флаг `MSG_PEEK` рекомендуется с особой аккуратностью, потому что вы можете встретиться с множеством непредсказуемых проблем. В этом случае функцию `recv` придется вызывать второй раз (без этого флага), чтобы удалить данные из системного буфера. При повторном считывании в буфере может оказаться больше данных, чем в первый раз (за это время компьютер может получить на порт дополнительные пакеты), и вы рискуете обработать данные дважды или не обработать что-то вообще. Еще одна проблема заключается в том, что системная память не очищается, и с каждым разом остается меньше пространства для поступающих данных. Именно поэтому я рекомендую использовать флаг `MSG_PEEK` только при необходимости и очень аккуратно.

Если функция отработала удачно, то она вернет количество байтов, полученных из сети. Обратите внимание, что именно байтов, а не символов. Если во время получения данных произошла ошибка по причине закрытия или обрыва соединения, то функция вернет 0 или –1.

Функция `WSARecv` выглядит следующим образом:

```
int WSARecv (
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
   );
```

Здесь также бросается в глаза сходство в параметрах с функцией `WSASend`. Давайте рассмотрим их назначение:

- s — сокет, через который будет происходить получение данных;
- lpBuffers — структура или массив структур типа WSABUF. В эти буферы будут помещены полученные данные;
- dwBufferCount — количество структур в параметре lpBuffers;
- lpNumberOfBytesSent — количество полученных байтов, если операции ввода/вывода уже завершились;
- dwFlags — определяет метод отправки и может принимать такие же значения, как и параметр dwFlags функции recv. Но есть один новый флаг — MSG_PARTIAL. Его нужно указывать для протоколов, ориентированных на чтение сообщения в несколько приемов. В случае указания этого флага при каждом считывании можно получить только часть данных;
- pOverlapped и pCompletionRoutine — устанавливаются при использовании перекрытого ввода/вывода (overlapped I/O). Это одна из моделей асинхронной работы сети, поддерживаемой WinSock.

Стоит заметить, что если вы используете протокол, ориентированный на передачу сообщений (UPD), и указали недостаточный размер буфера, то любая функция для получения данных вернет ошибку WSAEMSGSIZE. Если протокол потоковый (TCP), то такая ошибка не возникнет, потому что получаемые данные кэшируются в системе и предоставляются приложению полностью. В этом случае, если указан недостаточный буфер, то оставшиеся данные можно получить при следующем считывании.

Есть еще одна интересная сетевая функция, которая появилась в WinSock2. Если все рассмотренные в этой главе функции сетевой библиотеки (без префикса WSA) существуют не только в Windows, но и в UNIX-системах, то функция TransmitFile является расширением Microsoft и работает только в Windows.

Функция TransmitFile отсылает по сети целый файл. Это происходит достаточно быстро, потому что отправка идет через ядро библиотеки. Вам не надо заботиться о последовательном чтении и проверять количество отправленных данных, потому что это гарантируется библиотекой WinSock2.

Функция выглядит следующим образом:

```
BOOL TransmitFile(
    SOCKET hSocket,
    HANDLE hFile,
    DWORD nNumberOfBytesToWrite,
    DWORD nNumberOfBytesPerSend,
    LPOVERLAPPED lpOverlapped,
```

```
   LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
   DWORD dwFlags
  );
```

Рассмотрим ее параметры:

- `hSocket` — сокет, через который нужно отправить данные;
- `hFile` — указатель на открытый файл, данные которого надо отправить;
- `nNumberOfBytesToWrite` — количество отправляемых из файла байтов. Если указать `0`, то будет отправлен весь файл;
- `nNumberOfBytesPerSend` — размер пакета для отправки. Если указать `1024`, то данные будут отправляться пакетами в 1024 байтов данных. Если указать `0`, то будет использовано значение по умолчанию;
- `lpOverlapped` — используется при перекрестном вводе/выводе;
- `lpTransmitBuffers` — содержит служебную информацию, которую надо послать до и после отправки файла. По этим данным на принимающей стороне можно определить начало или окончание передачи;
- `dwFlags` — флаги. Здесь можно указать следующие значения:
  - `TF_DISCONNECT` — закрыть сокет после передачи данных;
  - `TF_REUSE_SOCKET` — подготовить сокет для повторного использования;
  - `TF_WRITE_BEHIND` — завершить работу, не дожидаясь подтверждения о получении данных со стороны клиента.

Параметр `lpTransmitBuffers` имеет тип структуры следующего вида:

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS;
```

У этой структуры следующие параметры:

- `Head` — указатель на буфер, содержащий данные, которые надо послать клиенту до начала отправки файла;
- `HeadLength` — размер буфера `Head`;
- `Tail` — указатель на буфер, содержащий данные, которые надо послать клиенту после завершения отправки файла;
- `TailLength` — размер буфера `Tail`.

4.6.7. Завершение соединения

Для завершения сеанса сначала необходимо проинформировать партнера, с которым происходило соединение, об окончании передачи данных. Для этого используется функция `shutdown`, которая выглядит следующим образом:

```
int shutdown (
    SOCKET s,
    int how
   );
```

Первый параметр — это сокет, соединение которого необходимо закрыть. Второй параметр может принимать одно из следующих значений:

- `SD_RECEIVE` — запретить любые функции приема данных. На протоколы нижнего уровня этот параметр не действует. Если используется потоковый протокол (например, TCP) и в очереди есть данные, ожидающие чтение функцией `recv`, или они пришли позже, то соединение сбрасывается. Если используется UDP-протокол, то сообщения продолжают поступать;
- `SD_SEND` — запретить все функции отправки данных;
- `SD_BOTH` — запретить прием и отправку данных.

После того как партнер проинформирован о завершении работы, можно закрывать сокет. Для этого используется функция `closesocket`, которая выглядит так:

```
int closesocket (
    SOCKET s
   );
```

После этого указанный в качестве единственного параметра сокет будет закрыт. Если вы попытаетесь использовать его в какой-нибудь функции, то получите ошибку `WSAENOTSOCK` — дескриптор не является сокетом. Любые пакеты, ожидающие отправки, прерываются или отменяются.

4.6.8. Принцип работы протоколов без установки соединения

Все описанное ранее относится к протоколам с установкой соединения между клиентом и сервером (протокол TCP), но существуют протоколы без установки соединения (например, UDP). Там не нужна функция `connect`, а прием и передача данных происходят по-другому. Я специально не затрагивал эту тему, чтобы вы не запутались в функциях и их назначении.

При работе с протоколами, не требующими соединения, на сервере достаточно вызвать функцию socket, чтобы связать сокет с портом и адресом (связать сокет и bind). После этого нельзя вызывать функции listen или accept, потому что сервер получает данные от клиента без установки соединения. Вместо этого нужно просто ожидать прихода данных с помощью функции recvfrom, которая выглядит следующим образом:

```
int recvfrom (
    SOCKET s,
    char FAR* buf,
    int len,
    int flags,
    struct sockaddr FAR* from,
    int FAR* fromlen
   );
```

Первые четыре параметра такие же, как и у функции recv. Параметр from указывает на структуру sockaddr, в которой будет храниться IP-адрес компьютера, с которого пришли данные. В параметре fromlen хранится размер структуры.

Во второй версии WinSock появилась функция WSARecvFrom, которая похожа на WSARecv, только добавлены параметры recv и fromlen:

```
int WSARecvFrom (
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
   );
```

С точки зрения клиента все тоже очень просто. Достаточно только создать сокет, и можно напрямую направлять данные. Для передачи данных по сети используется функция sendto:

```
int sendto (
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags,
    const struct sockaddr FAR * to,
```

```
    int tolen
   );
```

Первые четыре параметра соответствуют тем, что рассматривались в функции `send`. Параметр `to` — это структура типа `sockaddr`. Она содержит адрес и порт компьютера, которому нужно передать данные. Так как у нас нет соединения между клиентом и сервером, то эта информация должна указываться прямо в функции передачи данных. Последний параметр `tolen` — это размер структуры `to`.

Начиная со второй версии, мы можем пользоваться функцией `WSASendTo`. У нее параметры такие же, как и у `WSASend`, только добавлены два новых — `lpTo` и `iToLen`, хранящие соответственно структуру с адресом получателя и ее размер.

```
int WSASendTo (
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    const struct sockaddr FAR * lpTo,
    int iToLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
   );
```

Как видите, работа с протоколами, не требующими соединения, еще проще. Не надо вызывать функции прослушивания порта и соединения с сервером. Если вы разберетесь с работой протокола TCP, то работа UDP вам будет уже понятна.

При работе с UDP-протоколом нет ярко выраженного клиента и сервера, поэтому данная архитектура такой не является. Обе стороны получаются равноправными и поэтому на основе UDP часто строят распределенные системы типа чатов, обмен файлами. Если нужно передать файл, то задействуется TCP, но для посылки сообщений вполне подойдет протокол без установки соединений, который работает очень быстро и не требует установки в сети программы-сервера.

4.7. Примеры работы с сетью по протоколу TCP

Теперь пора на практике увидеть, как можно организовать работу в сети с помощью функций библиотеки WinSock. Для этого продемонстрирую не-

большую программу, в которой клиент будет посылать запросы серверу, а тот будет на них отвечать. На основе этого примера можно понять, как хакеры создают троянских коней и управляют или воруют данные с удаленного компьютера.

4.7.1. Пример работы TCP-сервера

Начну с разработки сервера. Создайте новый проект Win32 Project с именем TCPServer. Откройте файл TCPServer.cpp и после объявления всех глобальных переменных, но до функции `_tWinMain`, напишите две процедуры из листинга 4.12. Функции должны быть именно в таком порядке: сначала `ClientThread`, а затем — `NetThread`.

Листинг 4.12. Функции работы с сетью

```
DWORD WINAPI ClientThread(LPVOID lpParam)
{
    SOCKET   sock=(SOCKET)lpParam;
    char     szRecvBuff[1024],
       szSendBuff[1024];
    int      ret;
    // Запуск бесконечного цикла
    while(1)
    {
       // Получение данных
        ret = recv(sock, szRecvBuff, 1024, 0);
       // Проверка полученных данных
        if (ret == 0)
            break;
        else if (ret == SOCKET_ERROR)
        {
               MessageBox(0, "Recive data filed", "Error", 0);
            break;
        }
        szRecvBuff[ret] = '\0';

       // Здесь можно поставить проверку принятого текста
       // в переменной szRecvBuffer

       // Подготовка строки для отправки клиенту
       strcpy(szSendBuff, "Command get OK");

       // Отправка содержимого переменной szSendBuff клиенту
        ret = send(sock, szSendBuff, sizeof(szSendBuff), 0);
```

```
        if (ret == SOCKET_ERROR)
        {
           break;
        }
      }
    return 0;
}

DWORD WINAPI NetThread(LPVOID lpParam)
{
    SOCKET  sServerListen,
            sClient;
    struct sockaddr_in localaddr,
                       clientaddr;
    HANDLE  hThread;
    DWORD   dwThreadId;
    int     iSize;

    // Создание сокета
    sServerListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sServerListen == SOCKET_ERROR)
    {
       MessageBox(0, "Can't create socket", "Error", 0);
        return 0;
    }
    // Заполнение структуры localaddr типа sockaddr_in,
    // содержащей информацию о локальном адресе сервера и номер порта
    localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    localaddr.sin_family = AF_INET;
    localaddr.sin_port = htons(5050);

    // Связывание адреса с переменной localaddr типа sockaddr_in
    if (bind(sServerListen, (struct sockaddr *)&localaddr,
            sizeof(localaddr)) == SOCKET_ERROR)
    {
       MessageBox(0, "Can't bind", "Error", 0);
       return 1;
    }

    // Вывод сообщения об удачной операции bind
    MessageBox(0, "Bind OK", "Error", 0);

    // Запуск прослушивания порта
    listen(sServerListen, 4);
```

```
    // Вывод сообщения об удачном начале операции прослушивания
    MessageBox(0, "Listen OK", "Error", 0);

    // Запуск бесконечного цикла
    while (1)
    {
        iSize = sizeof(clientaddr);
        // Прием соединения из очереди. Если его нет,
        // то функция будет ожидать соединения клиента
        sClient = accept(sServerListen, (struct sockaddr *)&clientaddr,
                        &iSize);
        // Проверка корректности идентификатора клиентского сокета
        if (sClient == INVALID_SOCKET)
        {
            MessageBox(0, "Accept filed", "Error", 0);
            break;
        }

        // Создание нового потока для работы с клиентом
        hThread = CreateThread(NULL, 0, ClientThread,
                    (LPVOID)sClient, 0, &dwThreadId);
        if (hThread == NULL)
        {
            MessageBox(0, "Create thread filed", "Error", 0);
            break;
        }
        CloseHandle(hThread);
       }
       // Закрытие сокета после работы с потоком
       closesocket(sServerListen);
       return 0;
}
```

Теперь в функцию `_tWinMain`, до цикла обработки сообщений, добавьте следующий код:

```
WSADATA       wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
MessageBox(0, "Can't load WinSock", "Error", 0);
       return 0;
}

HANDLE        hNetThread;
DWORD         dwNetThreadId;
```

```
hNetThread = CreateThread(NULL, 0, NetThread,
                    0, 0, &dwNetThreadId);
```

Перейдем к рассмотрению написанного. В функции `_tWinMain` происходит загрузка библиотеки WinSock версии 2.2. В принципе, я ничего сверхъестественного не использовал, и можно было бы обойтись первой версией, но что мелочиться во времена Windows XP и Vista!

После этого создается новый поток с помощью функции `CreateThread`. Серверная функция `accept` блокирует работу программы, и если ее вызвать в основном потоке, то окно заблокируется и не будет реагировать на сообщения. Именно поэтому для сервера создается отдельный поток, в котором он и будет работать. Получается, что основная программа работает в своем потоке, а параллельно ей работает еще один поток, в котором сервер прослушивает необходимый порт.

С точки зрения программирования, поток — это функция, которая будет работать параллельно с другими потоками ОС и программой, создавшей поток. Таким образом, в ОС Windows реализуется многозадачность. За более подробной информацией о потоках обратитесь к документации или специализированной литературе по Visual C++.

В качестве третьего параметра функции `CreateThread`, создающей новый поток, необходимо передать указатель на функцию, которая должна работать в отдельном потоке.

Самое интересное происходит в функции `NetThread`. Все функции, которые там используются, мы уже рассмотрели, и здесь я только собрал все сказанное в одно целое.

Первым делом создается сокет функцией `socket`. Затем корректными параметрами заполняется структура `localaddr`, которая имеет тип `sockaddr_in`. Для предложенного сервера заполняются три параметра:

- `localaddr.sin_addr.s_addr` — указывается флаг `INADDR_ANY`, чтобы принимать подключения с любого интерфейса, установленного в компьютере;
- `localaddr.sin_family` — `AF_INET`, т. е. интернет-протоколы из семейства используемых протоколов;
- `localaddr.sin_port` — порт номер 5050. На большинстве компьютеров он свободен.

После этого связывается заполненная структура с сокетом с помощью функции `bind`.

Теперь сокет готов к началу прослушивания порта с помощью функции `listen`. В качестве второго параметра указано число 4, что соответствует очереди из четырех клиентов. Если одновременно попытаются подключиться

больше клиентов, то только первые четыре попадут в очередь, а остальные получат сообщение об ошибке.

Чтобы принимать соединения клиентов, запускается бесконечный цикл, в котором и будут обрабатываться все подключения. Почему бесконечный? Сервер должен всегда быть готовым принимать подключения от клиентов в любое время. Приняв одно подключение, он снова переходит в ожидание следующего.

Внутри цикла вызывается функция `accept`, чтобы принять соединение клиента из очереди. Как только соединение произошло, функция создаст сокет и вернет на него указатель, который сохраняется в переменной `sClient`. Прежде чем использовать новый сокет, его необходимо проверить на корректность. Если переменная `sSocket` будет равна `INVALID_SOCKET`, то с таким сокетом работать нельзя.

Если сокет корректный, то запускается еще один поток, в котором уже происходит обмен информацией (чтение данных, которые прислал клиент, ответы на запросы). Поток создается уже знакомой вам функцией `CreateThread`, а в качестве третьего параметра указывается функция `ClientThread`, которая и будет работать параллельно основной программе.

В качестве четвертого параметра функции `CreateThread` можно указывать любой параметр, и он будет передан функции потока. Логично будет указать клиентский сокет, чтобы в функции `ClientThread` знать сокет, с которым происходит работа.

В функции `ClientThread` передается только один параметр, в котором хранится то, что мы указали в качестве четвертого параметра при создании потока. В данном случае это дескриптор сокета, и первая строка кода функции дает этот сокет, который сохраняется в переменной `sock`:

```
SOCKET sock = (SOCKET)lpParam;
```

В этой функции также запускается бесконечный цикл на случай, если клиент будет присылать серверу множество команд, и на них надо будет отвечать. Внутри цикла сначала принимается текст с помощью функции `recv`. После этого полученные данные проверяются на корректность.

Если контроль прошел успешно, то можно проверить присланную клиентом команду. При написании троянского коня клиент может использовать запросы на высылку паролей, на перезагрузку компьютера, а может и запустить какую-нибудь шутку. Необходимо проверить, какой запрос пришел от сервера, и в зависимости от этого выполнить какие-либо действия.

Запросы могут приходить как простые текстовые команды, например, `restart` или `sendmepassword`. Так как мы только разбираем принцип действия

троянского коня, но не создаем его, то в примере клиент будет посылать текстовую команду `get`. Сервер же будет отсылать обратно клиенту текст `Command get OK`. Текст сохраняется в переменной, содержимое которой и отправляется клиенту с помощью функции `send`:

```
strcpy(szSendBuff, "Command get OK");

ret = send(sock, szSendBuff, sizeof(szSendBuff), 0);
if (ret == SOCKET_ERROR)
{
     break;
}
```

Затем цикл повторяется от начала, и если сервер считает еще одну команду, то он ответит на нее, иначе цикл прервется.

Как я уже говорил, все сетевые функции описаны в файле winsock2.h, и его необходимо подключать к своему проекту, чтобы компилятор не выдавал ошибок. В самом начале файла с исходным кодом нашей программы найдите следующею строку:

```
#include "stdafx.h"
```

После нее добавьте подключение модуля winsock2.h:

```
#include <winsock2.h>
```

Чтобы собрать проект без ошибок, необходимо подключить библиотеку ws2_32.lib. Для этого щелкните правой кнопкой мыши по имени проекта в окне **Solution Explorer** и в появившемся меню выберите пункт **Properties**.

Перед вами откроется окно свойств, в котором надо перейти в раздел **Configuration Properties/Linker/Input**. Здесь в строке **Additional Dependencies** напишите имя библиотеки ws2_32.lib.

Вот и все, что относится к серверной программе. Запустив ее, вы должны будете увидеть два сообщения: `Bind OK` и `Listen OK`. Если сообщения появились, то сервер работает корректно и находится в ожидании соединения со стороны клиента.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\TCPServer.

Но чтобы окончательно протестировать пример, надо написать программу клиента, который будет соединяться с сервером и отправлять ему команды. Именно это я покажу в следующем разделе.

4.7.2. Пример работы TCP-клиента

Сервер готов, теперь можно приступить к написанию клиентской части. Для этого создайте новый проект Win32 Project и назовите его TCPClient.

Найдите функцию `_tWinMain` и до цикла обработки сообщений добавьте следующий код:

```
WSADATA       wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
       MessageBox(0, "Can't load WinSock", "Error", 0);
       return 0;
}

HANDLE        hNetThread;
DWORD         dwNetThreadId;
hNetThread = CreateThread(NULL, 0, NetThread, 0, 0, &dwNetThreadId);
```

Здесь также загружается библиотека WinSock версии 2.2, хотя функции будут использоваться только из первой версии, и достаточно было бы ее.

Как и в случае с сервером, для работы с сетью будет использоваться отдельный поток, но для клиента достаточно только одного. Он также создается функцией `CreateThread`, а в качестве третьего параметра передается имя функции, которая будет выполняться в отдельном потоке — `NetThread`. Ее еще нет в созданном проекте, поэтому давайте введем сейчас. Добавьте до функции `_tWinMain` код из листинга 4.13.

Листинг 4.13. Поток работы с сетью

```
DWORD WINAPI NetThread(LPVOID lpParam)
{
    SOCKET  sClient;
    char    szBuffer[1024];
    int     ret, i;
    struct sockaddr_in server;
    struct hostent    *host = NULL;
    char  szServerName[1024], szMessage[1024];

    strcpy(szMessage, "get");
    strcpy(szServerName, "127.0.0.1");

    // Создание сокета
    sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
if (sClient == INVALID_SOCKET)
{
   MessageBox(0, "Can't create socket", "Error", 0);
   return 1;
}
// Заполнение структуры с адресом сервера и номером порта
server.sin_family = AF_INET;
server.sin_port = htons(5050);
server.sin_addr.s_addr = inet_addr(szServerName);

// Если указано имя, то перевод символьного адреса сервера в IP
if (server.sin_addr.s_addr == INADDR_NONE)
{
    host = gethostbyname(szServerName);
    if (host == NULL)
    {
       MessageBox(0, "Unable to resolve server", "Error", 0);
       return 1;
    }
    CopyMemory(&server.sin_addr, host->h_addr_list[0],
        host->h_length);
}
// Соединение с сервером
if (connect(sClient, (struct sockaddr *)&server,
    sizeof(server)) == SOCKET_ERROR)
{
    MessageBox(0, "connect failed", "Error", 0);
    return 1;
}

// Отправка и прием данных
ret = send(sClient, szMessage, strlen(szMessage), 0);
if (ret == SOCKET_ERROR)
{
   MessageBox(0, "send failed", "Error", 0);
}

// Задержка
Sleep(1000);

// Получение данных
char szRecvBuff[1024];
ret = recv(sClient, szRecvBuff, 1024, 0);
```

```
    if (ret == SOCKET_ERROR)
    {
       MessageBox(0, "recv failed", "Error", 0);
    }
    MessageBox(0, szRecvBuff, "Recived data", 0);
    closesocket(sClient);
}
```

Давайте подробно рассмотрим, что здесь происходит. В переменной `szMessage` хранится текст сообщения, которое отправляется серверу. Для примера жестко определена строка "`get`". В переменной `szServerName` указывается адрес сервера, с которым нужно произвести соединение. В данном случае установлен адрес 127.0.0.1, что соответствует локальному компьютеру. Это значит, что серверная и клиентская программы должны запуститься на одном и том же компьютере. После этого создается сокет так же, как и при создании сервера.

Следующим этапом надо подготовить структуру типа `sockaddr_in` (в нашем случае это структура `server`), в которой нужно указать семейство протоколов, порт (у сервера мы использовали 5050) и адрес сервера.

В примере указан IP-адрес, но в реальной программе у вас может быть и имя удаленного компьютера, которое нужно привести к IP. Именно поэтому адрес проверяется на равенство константе `INADDR_NONE`:

```
if (server.sin_addr.s_addr == INADDR_NONE)
```

Если условие выполняется, то в качестве адреса указано символьное имя, и тогда с помощью функции `gethostbyname` выполняется преобразование в IP-адрес. Результат записывается в переменную типа `hostent`. Как я уже говорил, компьютер может иметь несколько адресов, тогда результатом будет массив структур типа `hostent`. Чтобы не усложнять задачу, просто возьмите первый адрес, который можно получить так: `host->h_addr_list[0]`.

Теперь все готово к соединению с сервером. Для этого будет использоваться функция `connect`. Ей указывается созданный сокет, структура с адресом и размер структуры. Если функция вернет значение, отличное от `SOCKET_ERROR`, то соединение прошло успешно, иначе произошла ошибка.

Следующим этапом отправляются данные серверу с помощью функции `send`. Вроде бы все отправлено, и сервер должен ответить, но не стоит торопиться читать данные из буфера, потому что на передачу и обработку сервером информации нужно время. Если сразу после отправки попробовать вызвать функцию `recv`, то мы скорей всего получим ошибку, потому что данные еще не поступили. Именно поэтому после функции `send` нужно сделать задержку.

В реальной программе задержку делать не стоит, потому что можно поступить другим способом, например, запустить цикл получения сообщения и ожидать, пока функция `recv` вернет данные, а не ошибку. Это самый простой способ, который в данном случае будет работать корректно, если получаемые данные будут фрагментированы. В нашем тестовом примере мы точно знаем, что сервер даст только один ответ, и цикла не делаем.

Для компиляции проекта, как в случае с сервером, необходимо подключить модуль winsock2.h и библиотеку ws2_32.lib.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\TCPClient.

4.7.3. Анализ примера

Если сделать сервер невидимым и наделить его возможностями отправки паролей или перезагрузки компьютера по внешнему запросу, то этот пример легко превратить в "трояна". Но я не буду этого делать, чтобы не нарушить мои принципы. Все это рассматривалось только в познавательных целях.

Стоит также заметить, что после каждой операции при работе с сетью происходит проверка на ошибку. Если при создании сокета произошла какая-либо внештатная ситуация, то последующая работа бесполезна.

Давайте рассмотрим, как можно сделать описанный код более универсальным. В примере есть один недостаток. Если одна из сторон должна будет отправить данные слишком большого объема, то они будут отправлены/приняты не полностью. Это связано с тем, что данные уходят маленькими порциями (пакетами), и системный буфер для отправки данных не безграничен.

Допустим, что системный буфер равен 64 Кбайт. При попытке переслать по сети объем данных больше этого значения клиент получит только 64 Кбайт. Остальные данные просто пропадут. Чтобы этого не произошло, вы должны проверять, сколько реально было отправлено, и корректировать ваши действия.

В листинге 4.14 приведен пример, с помощью которого можно переслать клиенту любой объем данных, даже если он превышает размер буфера. Алгоритм достаточно прост, но давайте его подробно рассмотрим.

Листинг 4.14. Алгоритм отправки данных большого объема

```
char szBuff[4096];
szBuff = "Данные для отправки...";
```

```
int nSendSize = sizeof(szBuff);
int iCurrPos = 0;

while(nSendSize > 0)
{
       int ret = send(sock, &szBuff[iCurrPos], nSendSize, 0);
       if (ret == 0)
                break;
       else if (ret == SOCKET_ERROR)
              {
              // Произошла ошибка
              MessageBox(0, "Send failed", "Error", 0);
              break;
              }
       nSendSize -= ret;
       iCurrPos += ret;
}
```

В данном примере определяется размер отсылаемых данных и сохраняется в переменной `nSendSize`. После этого запускается цикл, который будет выполняться, пока переменная больше нуля и еще есть данные для отправки. Переменная `iCurrPos` указывает на текущую позицию в буфере, а отправка начинается с нулевой позиции.

В функции `send` в качестве второго параметра передается буфер, содержащий данные для отправки, а в квадратных скобках указана позиция в буфере, начиная с которой нужно отсылать.

Функция возвращает количество реально отосланных байтов. После проверки значения, которое вернула функция `send`, надо уменьшить размер данных в буфере, ожидающих отправки, и увеличить текущую позицию в буфере.

Тут нужно отметить еще одну особенность: возвращаемое значение сравнивается с нулем и константой `SOCKET_ERROR`. Если хотя бы одна из проверок прошла успешно, то соединение было прервано или завершено. Функция возвращает 0 в тех случаях, когда удаленная сторона завершила соединение закрытием сокета (`CloseSocket`). Если же произошла ошибка (результат равен `SOCKET_ERROR`), то соединение могло быть разорванным, и данных для обработки нет.

Самое страшное — не обработать ошибку и воспринять результат работы как целочисленное значение. Обратите внимание, что результат знаковый. Ошибка работы функции — это –1, и если ее воспринять как положительное число, то это будет FFFFFFFF. Это очень много данных, поэтому программа, скорей всего, рухнет.

Если отправлены еще не все данные, то на следующем шаге функция попытается отправить следующую порцию.

Вы также не сможете и принять сразу большую порцию данных — необходимо таким же образом запустить цикл. Но как определить, насколько велик этот кусок данных? Ведь при отправке известно количество данных, а при приеме — нет.

Решить эту проблему просто. Прежде чем отсылать данные, вы должны сообщить принимающей стороне количество байтов в пересылке. Для этого должен быть заведомо определен протокол передачи данных. Например, когда приходит команда `get`, то после нее определенное количество байтов можно отвести под значение размера отправляемых данных. Перед самими данными можно отправить команду `data`. Таким образом, клиент сможет получить их полностью. Код приема данных может выглядеть, как в листинге 4.15.

Листинг 4.15. Алгоритм получения данных большого объема

```
char szBuff[4096];
int nSendSize = 1000000;// Размер данных
int iCurrPos = 0;

while(nSendSize > 0)
{
       int ret = recv(sock, &szBuff[iCurrPos], nSendSize, 0);
       if (ret == 0)
              break;
       else if (ret == SOCKET_ERROR)
              {
              // Произошла ошибка
              MessageBox(0, "Send failed", "Error", 0);
              break;
              }
       nSendSize -= ret;
       iCurrPos += ret;
}
```

Если нужно передавать данные и в текстовом, и в бинарном виде, то можно поступить как в протоколе FTP. Там используются два канала связи: один для команд, а другой для данных. Когда на сервер приходит команда передачи бинарных данных, то сервер или клиент может открыть у себя порт для создания канала данных.

4.8. Примеры работы по протоколу UDP

Как вы уже могли понять, работа с протоколами, к каким относится UDP, происходит несколько иначе. Так как нет соединения между клиентом и сервером, то не нужно использовать некоторые функции, а точнее сказать, их использовать нельзя.

Функции, необходимые для работы с протоколом UDP, я уже описал в *разд. 4.6.8*, и теперь вам предстоит увидеть реальный пример и применить полученные знания на практике.

4.8.1. Пример работы UDP-сервера

Создайте новый проект Win32 Project и назовите его UDP Server. Откройте файл UDPServer.cpp и добавьте в функцию `_tWinMain` перед циклом обработки сообщений следующий код:

```
WSADATA       wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    MessageBox(0, "Can't load WinSock", "Error", 0);
    return 0;
}

HANDLE        hNetThread;
DWORD         dwNetThreadId;
hNetThread = CreateThread(NULL, 0, NetThread, 0, 0, &dwNetThreadId);
```

Как и в случае с TCP-сервером, загружается библиотека WinSock и создается новый поток, в котором и будет происходить работа с сетью. Сама функция потока показана в листинге 4.16.

Листинг 4.16. Функция работы с сетью

```
DWORD WINAPI NetThread(LPVOID lpParam)
{
       SOCKET         sServerListen;
       struct sockaddr_in localaddr,
                      clientaddr;
       int            iSize;

       sServerListen = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sServerListen == INVALID_SOCKET)
```

```
        {
                MessageBox(0, "Can't create socket", "Error", 0);
                return 0;
        }
        localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        localaddr.sin_family = AF_INET;
        localaddr.sin_port = htons(5050);

        if (bind(sServerListen, (struct sockaddr *)&localaddr,
             sizeof(localaddr)) == SOCKET_ERROR)
        {
                MessageBox(0, "Can't bind", "Error", 0);
                return 1;
        }

        MessageBox(0, "Bind OK", "Warning", 0);

        char buf[1024];

        while (1)
        {
                iSize = sizeof(clientaddr);
         int ret = recvfrom(sServerListen, buf, 1024, 0,
                        (struct sockaddr *)&clientaddr, &iSize);
                MessageBox(0, buf, "Warning", 0);
    }
    closesocket(sServerListen);
    return 0;
}
```

Во время создания сокета функцией `socket` указывается параметр `SOCK_DGRAM`, что означает необходимость использования протокола, основанного на сообщениях. В качестве последнего параметра нужно указать константу, точно определяющую протокол. В данном случае можно явно указать UDP-протокол с помощью константы `IPPROTO_UDP` или просто указать значение `0`.

Все остальное вам уже должно быть понятно. После создания сокета нужно привязать его к локальному адресу функцией `bind`. Для UDP-сервера этого достаточно. В примере после связывания сокета запускается бесконечный цикл, который вызывает функцию `recvfrom` для получения данных от клиента.

При получении данных сервер просто выводит на экран окно с полученной информацией. Адрес отправителя сохраняется в переменной `clientaddr`, и его можно использовать для отправки ответа клиенту.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\UDPServer.

4.8.2. Пример работы UDP-клиента

Теперь опишу программу клиента, который будет отправлять данные на сервер. Создайте новый проект Win32 Project и назовите его UDPClient. В данном случае можно обойтись без дополнительных потоков и отправить данные прямо из функции `_tWinMain`. Это связано с тем, что передача по протоколу UDP не делает задержек, и данные могут отправляться практически мгновенно. Поэтому не имеет смысла делать многозадачное приложение, что значительно упрощает задачу.

Откройте файл UDPClient.cpp и перед циклом обработки событий напишите код из листинга 4.17.

Листинг 4.17. Отправка данных UDP-серверу

```
WSADATA       wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
       MessageBox(0, "Can't load WinSock", "Error", 0);
       return 0;
}

SOCKET        sSocket;
struct sockaddr_in servaddr;
char  szServerName[1024], szMessage[1024];
struct hostent    *host = NULL;

strcpy(szMessage, "This is message from client");
strcpy(szServerName, "127.0.0.1");

sSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sSocket == INVALID_SOCKET)
{
       MessageBox(0, "Can't create socket", "Error", 0);
       return 0;
}
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(5050);
servaddr.sin_addr.s_addr = inet_addr(szServerName);
```

```
if (servaddr.sin_addr.s_addr == INADDR_NONE)
{
       host = gethostbyname(szServerName);
       if (host == NULL)
       {
              MessageBox(0, "Unable to resolve server", "Error", 0);
              return 1;
       }
       CopyMemory(&servaddr.sin_addr, host->h_addr_list[0],
       host->h_length);
}

sendto(sSocket, szMessage, 30, 0, (struct sockaddr *)&servaddr,
              sizeof(servaddr));
```

Как и в случае с сервером, необходимо создать сокет, где в качестве второго параметра указано значение `SOCK_DGRAM`. Третий параметр определяет протокол, в данном случае это `IPPROTO_UDP`.

После этого заполняется переменная `servaddr` типа `sockaddr_in`, которая содержит адрес и порт компьютера, которому нужно отправить данные. Если в качестве адреса указано символьное имя, то оно преобразуется в IP-адрес так же, как и при TCP-клиенте.

Теперь можно напрямую без соединения с сервером отправлять данные функцией `sendto`. В данном случае серверу отправляется содержимое переменной `szMessage`.

Не забывайте, что для компиляции примеров, использующих работу с сетью, необходимо подключить библиотеку ws2_32.lib *(см. разд. 4.7.1)*.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\UDPClient.

4.9. Обработка принимаемых данных

Все принятые по сети данные следует тщательно верифицировать. Если необходимо разделить доступ к определенным возможностям по паролю, то я рекомендую первым делом контролировать права на выполнение команды. После этого проверяйте корректность указанной команды и переданные параметры.

Допустим, что клиент запрашивает у сервера какие-либо файлы. Если вы сначала проверите правильность указания пути и имени файла и в случае не-

удачи выведете сообщение об ошибке, то хакер будет знать, что файла в системе нет. Иногда хакеру этой информации может оказаться достаточно для поиска пути проникновения в систему. Именно поэтому сначала нужно проверять право на выполнение команды, а потом уже разбирать параметры и оценивать их корректность.

Если есть возможность, то команды лучше проверять жестко. Например, команду `get filename` необходимо проверять так, чтобы первые три буквы составляли слово "get". Нельзя делать поиск символов "get" во всем полученном тексте, т. к. для хакера открывается множество возможностей передать неверные данные. Большинство взломов происходит из-за неправильного анализа полученных данных и передачи некорректных данных серверу.

Если вы разрабатываете протокол обмена командами между клиентом и сервером, то делайте так, чтобы команда передавалась в самом начале, а все параметры шли в конце. Допустим, что у вас есть команда `get`. Она может работать в двух режимах:

- `GET Имя файла FROM Адрес` — забрать файл от стороннего сервера;
- `GET Имя файла` — забрать файл от клиента, который подключился к серверу.

Первая команда, с точки зрения безопасности, неэффективна. Для определения наличия ключевого слова `FROM` придется делать поиск по строке. Этого делать нельзя. Все ключевые слова желательно искать в жестко определенной позиции. В данном случае первую команду желательно преобразовать к следующему виду:

```
GET FROM Имя файла, Адрес
```

В этом случае ключевые слова идут в начале команды, и вы жестко можете определить их наличие. Если хакер попытается использовать неправильные параметры, то у него ничего не выйдет.

Имя файла не может иметь определенного размера или формата. Одно имя может быть из трех букв и не иметь пробелов, а другое может быть из 100 букв с пробелами, точками и другими символами. Поэтому имена лучше указывать в самом конце, а для большей надежности желательно обрамлять двойными кавычками. Таким образом, будет проще его вырезать.

Если передаваемые данные разнообразны, но могут быть подведены под какой-то шаблон, то обязательно используйте его при контроле. Это поможет вам сделать дополнительную проверку корректности данных, но не может обеспечить полную защиту. Именно в шаблонной проверке программисты чаще всего допускают ошибки. Чем сложнее проверка или шаблон, тем труднее учесть все ограничения на передаваемые данные. Прежде чем использовать программу в боевых условиях или в коммерческих целях, рекомендуется

уделить тестированию этого участка максимально возможное время. Желательно, чтобы программу тестировал сторонний человек (и профессионал в данной области), потому что только конечный пользователь или хакер введет те данные, о которых вы даже не подозревали и не думали, что их вообще можно использовать.

Задача усложняется, если по этим данным будет происходить доступ к файловой системе. Это может привести к нерегламентированному доступу к вашему диску со всеми вытекающими последствиями. Когда в качестве параметра указывается путь к файлу, то его легко проверить по шаблону, но очень легко ошибиться. Большинство программистов просто проверяют начало пути, но это ошибка.

Допустим, что у вас открыт доступ только к папке interpub на диске C:. Если проверять только начало пути, то хакер сможет без проблем написать вот такой путь:

```
c:\interpub\..\winnt\system32\cmd.exe\
```

Здесь, благодаря двойной точке, хакер выходит из папки interpub и получает доступ ко всему диску, в том числе и системным файлам.

Прежде чем писать проверку по шаблону, вы должны ознакомиться со всеми его исключительными ситуациями. И еще раз напоминаю, что вы должны максимально тестировать программу даже с самыми невероятными параметрами. Пользователи непредсказуемы, особенно неопытные, а хакеры достаточно умны и изучают систему со всех сторон, даже с тех, о которых вы не подозреваете.

4.10. Прием и передача данных

Вы уже познакомились в теории и на практике, как принимать и передавать данные между компьютерами. Но искусство хакера состоит в том, чтобы правильно использовать различные методы и режимы передачи. Существует два режима работы сокетов, и вы должны научиться правильно их использовать, потому что это повысит эффективность и скорость ваших программ.

Применяют следующие режимы сетевого ввода/вывода (прием/передача данных):

- *блокирующий* (синхронный) — при вызове функции передачи программа останавливает выполнение и ожидает завершения операции;
- *не блокирующий* (асинхронный) — после вызова функции программа продолжает выполнение вне зависимости от того, закончена операция приема/передачи или нет.

При описании функций мы уже сталкивались с этими понятиями *(см. разд. 4.6.5 и 4.6.6)*, а сейчас остановлюсь на них более подробно, чтобы значительно повысить скорость работы и максимально использовать ресурсы.

По умолчанию создаются блокирующие сокеты, поэтому во всех примерах, которые уже рассматривались в этой главе, использовался синхронный режим — как наиболее простой. В этом случае приходится создавать потоки, внутри которых работают сетевые функции, чтобы главное окно программы не блокировалось и реагировало на события от пользователя.

Но это не самая главная проблема. Простота и надежность — не всегда совместимые понятия. Допустим, что был вызов функции `recv`, но по каким-то причинам она не вернула данные. В этом случае она останется заблокированной надолго, и сервер больше не будет реагировать на действия пользователя. Некоторые программисты перед считыванием данных проверяют их корректность с помощью вызова функции `recv` с флагом `MSG_PEEK`. Но вы уже знаете, что это небезопасно, и доверять таким данным не стоит. К тому же этот метод нагружает систему лишними проверками буфера приема на наличие данных.

Не блокирующие сокеты сложнее в программировании, но лишены описанных недостатков. Чтобы перевести сокет в асинхронный режим, нужно воспользоваться функцией `ioctlsocket`, которая выглядит так:

```
int ioctlsocket (
    SOCKET s,
    long cmd,
    u_long FAR* argp
   );
```

У этой функции три параметра:

- сокет, режим которого надо изменить;
- команда, которую необходимо выполнить;
- параметр для команды.

Изменение режима блокирования происходит при указании в качестве команды константы `FIONBIO`. При этом если параметр команды имеет нулевое значение, то будет использоваться блокирующий режим, иначе — не блокирующий.

Давайте посмотрим на пример создания сокета и перевода его в не блокирующий режим:

```
SOCKET s;
unsigned long ulMode;
```

```
s = socket(AS_INET, SOCK_STREAM, 0);
ulMode = 1;
ioctlsocket(s, FIONBIO, (unsigned long*)&ulMode);
```

Теперь все функции приема/передачи будут завершаться ошибкой. Это нормальная реакция, и вы должны это учитывать при создании сетевых приложений, работающих в не блокирующем режиме. Если функция ввода/вывода вернула ошибку `WSAEWOULDBLOCK`, то это не означает неправильную передачу. Все прошло успешно. Если же действительно произошел сбой, то мы получим ошибку, отличную от `WSAEWOULDBLOCK`.

В асинхронном режиме функция `recv` не будет дожидаться приема данных, а просто вернет ошибку `WSAEWOULDBLOCK`. Тогда как нам узнать, что данные поступили на порт? Некоторые запускают цикл с постоянным вызовом функции `recv`, пока она не вернет данные. Но это нецелесообразно, потому что происходит блокирование приложения и излишне занят процессор.

Конечно же, вы можете в цикле между проверками выполнять какие-то действия и тем самым использовать процессор во время ожидания с пользой, но я не буду рассматривать этот вариант, потому что есть способ лучше.

4.10.1. Функция *select*

Еще в первой версии Winsock была очень интересная возможность управления неблокируемыми сокетами. Для этого используется функция `select`:

```
int select (
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout
   );
```

Функция возвращает количество готовых к использованию дескрипторов Socket.

Теперь рассмотрим параметры этой функции:

- `nfds` — игнорируется и служит только для совместимости с моделью сокетов Беркли;
- `readfds` — возможность чтения (структура типа `fd_set`);
- `writefds` — возможность записи (структура типа `fd_set`);
- `exceptfds` — важность сообщения (структура типа `fd_set`);
- `timeout` — максимальное время ожидания или `NULL` для блокирования дальнейшей работы (ожидать бесконечно).

Структура fd_set — набор сокетов, от которых нужно ожидать разрешение на выполнение определенной операции. Например, если вам нужно дождаться прихода данных на один из двух сокетов, то вы можете сделать следующее:

- добавить в набор fd_set два уже созданных сокета;
- запустить функцию select и в качестве второго параметра указать набор с сокетами.

Функция select будет ожидать данные указанное время, после чего можно прочитать данные из сокета. Но данные могут прийти только на один из двух сокетов. Как узнать, на какой именно? Для начала с помощью функции FD_ISSET нужно обязательно проверить, входит ли сокет в набор.

При работе со структурой типа fd_set вам понадобятся следующие функции:

- FD_ZERO — очищает набор. Прежде чем добавлять в набор новые сокеты, обязательно вызывайте эту функцию, чтобы проинициализировать набор. У этой функции только один параметр — указатель на переменную типа fd_set;
- FD_SET — добавляет сокет в набор. У функции два параметра — сокет, который нужно добавить, и переменная типа fd_set, в набор которой нужно добавить сокет;
- FD_CLR — удаляет сокет из набора. У этой функции два параметра — сокет, который надо удалить, и набор, из которого будет происходить удаление;
- FD_ISSET — проверяет, входит ли сокет, определенный в первом параметре, в набор типа fd_set, указанный в качестве второго параметра.

4.10.2. Простой пример использования функции *select*

Теперь применим все сказанное на практике. Откройте пример TCPServer из *разд. 4.7.1* и после создания сокета добавьте следующий код:

```
ULONG ulBlock;
ulBlock = 1;
if (ioctlsocket(sServerListen, FIONBIO, &ulBlock) == SOCKET_ERROR)
{
   return 0;
}
```

Таким образом сокет переводится в асинхронный режим. Теперь попробуйте запустить пример. Сначала вы должны увидеть два сообщения: Bind OK и

`Listen OK`, после чего программа вернет ошибку `Accept filed`. В асинхронном режиме функция `accept` не блокирует работу программы, а значит, не ожидает соединения. В этом случае, если в очереди нет ожидающих подключения клиентов, то функция вернет ошибку `WSAEWOULDBLOCK`.

Чтобы избавиться от этого недостатка, нужно подкорректировать цикл ожидания соединения (бесконечный цикл `while`, который идет после вызова функции `listen`). Для асинхронного варианта он должен выглядеть, как в листинге 4.18.

Листинг 4.18. Цикл ожидания соединения

```
FD_SET ReadSet;
int ReadySock;

while (1)
{
   FD_ZERO(&ReadSet);
   FD_SET(sServerListen, &ReadSet);

   if ((ReadySock = select(0, &ReadSet, NULL, NULL, NULL)) ==
              SOCKET_ERROR)
   {
       MessageBox(0, "Select filed", "Error", 0);
   }

   if (FD_ISSET(sServerListen, &ReadSet))
   {
       iSize = sizeof(clientaddr);
       sClient = accept(sServerListen,
             (struct sockaddr *)&clientaddr, &iSize);
       if (sClient == INVALID_SOCKET)
       {
           MessageBox(0, "Accept filed", "Error", 0);
           break;
       }

       hThread = CreateThread(NULL, 0, ClientThread,
                            (LPVOID)sClient, 0, &dwThreadId);
       if (hThread == NULL)
       {
          MessageBox(0, " thread filed", "Error", 0);
          break;
       }
```

```
        CloseHandle(hThread);
    }
}
```

Перед циклом добавлены две переменные: `ReadSet` типа `FD_SET` для хранения набора сокетов и `ReadySock` типа `int` для хранения количества готовых к использованию сокетов. На данный момент у нас только один сокет, поэтому эту переменную пока использовать не будем.

В самом начале цикла обнуляется набор с помощью функции `FD_ZERO` и добавляется созданный сокет, который ожидает подключения. После этого вызывается функция `select`. Для нее указан только второй параметр, а все остальные значения — нулевые. Если указан второй параметр, то функция ожидает возможности чтения для сокетов из набора. Параметр "время ожидания" тоже установлен в ноль, что соответствует бесконечному ожиданию.

Итак, сокет сервера ожидает подключения и готов к чтению. Когда от клиента поступит запрос на подключение, сокет примет его. Но прежде чем выполнять какие-то действия, необходимо проверить вхождение сокета в набор с помощью функции `FD_ISSET`.

Остальной код не изменился. Мы принимаем входящее соединение с помощью функции `accept`, получаем новый сокет для работы с клиентом и сохраняем его в переменной `sClient`. После этого создается новый поток, в котором происходит обмен данными с клиентом.

Запустите пример и убедитесь, что он работает корректно. Теперь нет ошибок, и программа терпеливо ожидает соединения со стороны клиента.

Возникает вопрос, в каком режиме работает сокет `sClient`, который создан функцией `accept`. Я уже говорил, что по умолчанию сокеты работают в блокирующем режиме, и мы не изменяли это значение. Точно так же поступают сокеты, созданные для работы с удаленным клиентом. Они также по умолчанию будут работать в блокирующем режиме или для выбора иного значения нужно изменить на не блокирующий.

С помощью функции `select` можно избавиться от второго потока, который используется для обмена данными между клиентом и сервером. Помимо этого, в примере в нынешнем виде для обработки нескольких клиентов нужно создавать множество потоков. Благодаря функции `select` можно все это сделать без потоков, намного проще и эффективнее. Но к этому я вернусь в *главе 6*, где будут рассматриваться интересные алгоритмы.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\Select.

4.10.3. Использование сокетов через события Windows

Функция `select` введена в библиотеку WinSock для совместимости с аналогичными библиотеками других платформ. Для программирования в Windows более мощной является функция `WSAAsyncSelect`, которая позволяет отслеживать состояние сокетов с помощью сообщений Windows. Таким образом, вы сможете получать сообщения в функции `WndProc`, и нет необходимости замораживать работу программы для ожидания доступности сокетов.

Функция выглядит следующим образом:

```
int WSAAsyncSelect (
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
   );
```

Рассмотрим каждый параметр:

- `s` — сокет, события которого необходимо ловить;
- `hWnd` — окно, которому будут посылаться события при возникновении сетевых сообщений. Именно у этого окна (или родительского) должна быть функция `WndProc`, которая будет получать сообщения;
- `wMsg` — сообщение, которое будет отсылаться окну. По его типу можно определить, что это событие сети;
- `lEvents` — битовая маска сетевых событий, которые нас интересуют. Этот параметр может принимать любую комбинацию из следующих значений:
  - `FD_READ` — готовность к чтению;
  - `FD_WRITE` — готовность к записи;
  - `FD_OOB` — получение срочных данных;
  - `FD_ACCEPT` — подключение клиентов;
  - `FD_CONNECT` — соединение с сервером;
  - `FD_CLOSE` — закрытие соединения;
  - `FD_QOS` — изменения сервиса QoS (Quality of Service);
  - `FD_GROUP_QOS` — изменение группы QoS.

Если функция отработала успешно, то она вернет значение больше нуля, если произошла ошибка — `SOCKET_ERROR`.

Функция автоматически переводит сокет в не блокирующий режим, и нет смысла вызывать функцию `ioctlsocket`.

Вот простой пример использования `WSAAsyncSelect`:

```
WSAAsyncSelect(s, hWnd, wMsg, FD_READ|FD_WRITE);
```

После выполнения этой строчки кода окно `hWnd` будет получать событие `wMsg` каждый раз, когда сокет `s` будет готов принимать и отправлять данные. Чтобы отменить работу события, необходимо вызвать эту же функцию, но в качестве четвертого параметра указать 0:

```
WSAAsyncSelect(s, hWnd, 0, 0);
```

В данном случае необходимо правильно указать первые два параметра и обнулить последний. Содержимое третьего параметра не имеет значения, потому что событие не будет отправляться, и можно указать ноль. Если вам нужно просто изменить типы событий, то можете вызвать функцию с новыми значениями четвертого параметра. Нет смысла сначала обнулять, а потом устанавливать заново.

Для каждого сокета можно назначить только одно сообщение на разные события. Это означает, что нельзя по событию `FD_READ` окну посылать одно сообщение, а по `FD_WRITE` — другое, но это не проблема. Обработчик события у окна один, и его вполне достаточно.

Прежде чем приступать к рассмотрению примера, надо разобраться с параметрами, которые будут передаваться в функцию `WndProc` при возникновении события. Вспомним, как выглядит эта функция:

```
LRESULT CALLBACK WndProc(
 HWND hWnd,
 UINT message,
 WPARAM wParam,
 LPARAM lParam
)
```

Параметры `wParam` и `lParam` содержат вспомогательную информацию (в зависимости от события). Для событий сети в параметре `wParam` хранится дескриптор сокета, на котором произошло событие. Таким образом, вам не надо хранить массив созданных сокетов, их всегда можно получить в событии, где и будет находиться вся логика работы с удаленным клиентом.

Параметр `lParam` состоит из двух слов: младшее определяет событие, а старшее — код ошибки. Вот теперь можно переходить к рассмотрению реального примера. Создайте новое приложение Win32 Application, назовите проект WSASel. Откройте файл WSASel.cpp и подкорректируйте функцию `_tWinMain`.

Как всегда, весь код нужно добавить до цикла обработки сообщений. Всю функцию вы можете увидеть в листинге 4.19.

Листинг 4.19. Функция `_tWinMain`

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
        // TODO: Place code here.
       MSG msg;
       HACCEL hAccelTable;

       // Initialize global strings
       LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
       LoadString(hInstance, IDC_WSASEL, szWindowClass, MAX_LOADSTRING);
       MyRegisterClass(hInstance);

       // Perform application initialization:
       if (!InitInstance (hInstance, nCmdShow))
              return FALSE;

       hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_WSASEL);

       WSADATA       wsd;
       if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
       {
              MessageBox(0, "Can't load WinSock", "Error", 0);
              return 0;
       }

       SOCKET sServerListen, sClient;
       struct sockaddr_in localaddr, clientaddr;
       HANDLE hThread;
       DWORD dwThreadId;
       int iSize;

       sServerListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
       if (sServerListen == SOCKET_ERROR)
       {
              MessageBox(0, "Can't load WinSock", "Error", 0);
              return 0;
       }
```

```
    ULONG ulBlock;
    ulBlock = 1;
    if (ioctlsocket(sServerListen, FIONBIO, &ulBlock) == SOCKET_ERROR)
        return 0;

    localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    localaddr.sin_family = AF_INET;
    localaddr.sin_port = htons(5050);

    if (bind(sServerListen, (struct sockaddr *)&localaddr,
         sizeof(localaddr)) == SOCKET_ERROR)
    {
        MessageBox(0, "Can't bind", "Error", 0);
        return 1;
    }

    WSAAsyncSelect(sServerListen, hWnd, WM_USER+1, FD_ACCEPT);
    listen(sServerListen, 4);

    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    closesocket(sServerListen);
    WSACleanup();

    return (int) msg.wParam;
}
```

Благодаря использованию функции `WSAAsyncSelect` весь код (без дополнительных потоков) можно расположить прямо в функции `_tWinMain`.

Код практически ничем не отличается от того, что был в проекте TCPServer *(см. разд. 4.7.1)*. Единственное: перед запуском прослушивания (`listen`) вызывается функция `WSAAsyncSelect`, чтобы выбрать созданный сокет и перевести его в асинхронный режим. Здесь указываются следующие параметры:

- `sServerListen` — переменная, которая указывает на созданный серверный сокет;

- `hWnd` — указатель на главное окно программы, и именно ему будут передаваться сообщения;
- `WM_USER+1` — все пользовательские сообщения должны быть больше константы `WM_USER`. Меньшие значения могут использоваться системой и вызвать конфликт. Я использовал такую конструкцию, чтобы явно показать необходимость использования такого сообщения. В реальных приложениях я советую создавать для этого константу с понятным именем и использовать ее. Это можно сделать следующим образом: `#define WM_NETMESSAGE WM_USER+1`;
- `FD_ACCEPT` — событие, которое нужно обрабатывать. Что может делать серверный сокет? Принимать соединения со стороны клиента. Именно это событие нас интересует.

Самое главное будет происходить в функции `WndProc`. Начало функции, где нужно добавить код, показано в листинге 4.20.

Листинг 4.20. Обработка сетевых сообщений в функции `WndProc`

```
LRESULT CALLBACK WndProc(HWND hWnd,
UINT message, WPARAM wParam, LPARAM lParam)
{
       int wmId, wmEvent;
       PAINTSTRUCT ps;
       HDC hdc;

       SOCKET ClientSocket;
       int ret;
       char szRecvBuff[1024], szSendBuff[1024];

       switch (message)
       {
       case WM_USER+1:
              switch (WSAGETSELECTEVENT(lParam))
              {
              case FD_ACCEPT:
                     ClientSocket = accept(wParam, 0, 0);
                     WSAAsyncSelect(ClientSocket, hWnd, WM_USER+1,
                         FD_READ | FD_WRITE | FD_CLOSE);
                     break;

              case FD_READ:
                     ret = recv(wParam, szRecvBuff, 1024, 0);
```

```
                    if (ret == 0)
                  break;
                    else if (ret == SOCKET_ERROR)
                     {
                    MessageBox(0, "Recive data filed",
                               "Error", 0);
                  break;
                    }
                    szRecvBuff[ret] = '\0';

                    strcpy(szSendBuff, "Command get OK");

                    ret = send(wParam, szSendBuff,
                          sizeof(szSendBuff), 0);
                    break;

             case FD_WRITE:
                    //Ready to send data (Готов к передаче данных)
                    break;

             case FD_CLOSE:
                    closesocket(wParam);
           break;
        }
        case WM_COMMAND:
...
```

Здесь в самом начале добавлен новый оператор `case`, который проверяет, равно ли пойманное сообщение искомому сетевому сообщению `WM_USER+1`. Если это сетевое событие, то запускается перебор сетевых событий. Для этого используется оператор `switch`, который сравнивает указанное в скобках значение с поступающими событиями:

```
switch (WSAGETSELECTEVENT(lParam))
```

Как известно, в параметре `lParam` находятся код ошибки и тип события. Чтобы получить событие, используется функция `WSAGETSELECTEVENT`. А затем проверяются необходимые нам события. Если произошло соединение со стороны клиента, то выполняется следующий код:

```
case FD_ACCEPT:
ClientSocket = accept(wParam, 0, 0);
WSAAsyncSelect(ClientSocket, hWnd, WM_USER+1,
FD_READ | FD_WRITE | FD_CLOSE);
break;
```

Сначала принимается соединение с помощью функции `accept`. Результатом будет сокет, с помощью которого можно работать с клиентом. С этого сокета тоже нужно ловить события, поэтому вызываем функцию `WSAAsyncSelect`. Чтобы не плодить сообщения, используем в качестве третьего параметра значение `WM_USER+1`. Это не вызовет конфликтов, потому что серверный сокет обрабатывает только событие `FD_ACCEPT`, а у клиентского нас интересуют события чтения, записи данных и закрытия сокета, так что никаких перекрещений не будет. Если же серверный сокет будет требовать таких же событий, что и клиент, то их события можно разделить, например, назначив клиенту событие `WM_USER+2`.

Когда к серверу придут данные, поступит сообщение `WM_USER+1`, а функция `WSAGETSELECTEVENT(lParam)` вернет значение `FD_READ`. В этом случае читаются пришедшие данные, а клиенту посылается текст "Command get OK":

```
case FD_READ:
       ret = recv(wParam, szRecvBuff, 1024, 0);
       if (ret == 0)
                break;
             else if (ret == SOCKET_ERROR)
            {
                MessageBox(0, "Recive data filed", "Error", 0);
                break;
            }
            szRecvBuff[ret] = '\0';
            strcpy(szSendBuff, "Command get OK");
            ret = send(wParam, szSendBuff, sizeof(szSendBuff), 0);
       break;
```

Это тот же самый код, который использовался в приложении TCPServer для обмена данными между клиентом и сервером. Я намеренно не вносил изменений, чтобы сервер можно было протестировать программой TCPClient.

По событию `FD_WRITE` ничего не происходит, а только стоит комментарий. По событию `FD_CLOSE` закрывается сокет.

Рассмотренный ранее пример с использованием функции `select` может работать только с одним клиентом. Чтобы добавить возможность одновременной обработки нескольких соединений, необходимо сформировать массив потоков, используемых для приема/передачи данных. В *главе 6* я приведу пример с использованием функции `select`, который и без массивов потоков будет лишен этих недостатков.

Функция `WSAAsyncSelect` проще в программировании и изначально позволяет обрабатывать множество клиентов, как в данном примере. Ну, а самое главное — нет ни одного дополнительного потока.

Чтобы протестировать пример, сначала запустите программу-сервер WSASel, а потом — программу-клиент TCPClient.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter4\WSASel.

Хочу обратить ваше внимание, что обмен информацией происходит асинхронно. Отправку и прием большого количества данных нужно будет делать порциями.

Допустим, что клиент должен передать серверу 1 Мбайт данных. Конечно же, за один прием это сделать нереально. Поэтому на стороне сервера вы должны действовать следующим образом:

1. Сервер должен узнать (клиент должен сообщить серверу) количество передаваемых данных.
2. Сервер должен выделить необходимый объем памяти или, если количество данных слишком большое, создать временный файл.
3. При получении события `FD_READ` следует сохранять принятые данные в буфере или в файле. Далее нужно обрабатывать событие, пока данные не будут получены полностью и клиент не пришлет определенную последовательность байтов, определяющую завершение передачи данных.

Подобным способом должна происходить отправка от клиента:

1. Сообщить серверу количество отправляемых данных.
2. Открыть файл, из которого будут читаться данные.
3. Послать первую порцию данных, остальные данные — по событию `FD_WRITE`.
4. По окончании отправки послать серверу последовательность байтов, определяющую завершение передачи данных.

Использование сообщений Windows очень удобно, но вы теряете совместимость с UNIX-системами, где сообщения реализованы по-другому и нет функции `WSAAsyncSelect`. Поэтому при переносе такой программы на другую платформу возникнут большие проблемы и придется переписать слишком много кода. Но если перенос не планируется, то я всегда использую `WSAAsyncSelect`, что позволяет добиться максимальной производительности и удобства программирования.

4.10.4. Асинхронная работа через объект события

Если в программе нет процедуры обработки сообщений, то можно воспользоваться объектами событий. В этом случае алгоритм работы будет несколько иной.

1. Создать объект события с помощью функции `WSACreateEvent`.
2. Выбрать сокет с помощью функции `WSAEventSelect`.
3. Ожидать событие с помощью функции `WSAWaitForMultipleEvents`.

Давайте подробно рассмотрим все функции, необходимые для работы с объектами событий.

Первым делом следует создать событие с помощью функции `WSACreateEvent`. Функции не надо передавать никаких параметров, она просто возвращает новое событие типа `WSAEVENT`:

```
WSAEVENT  WSACreateEvent(void);
```

Теперь нужно связать сокет с этим объектом и указать события, которые нам нужны. Для этого используется функция `WSAEventSelect`:

```
int WSAEventSelect (
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
)
```

Первый параметр — это сокет, события которого нас интересуют. Второй параметр — объект события. Последний параметр — это необходимые события. В качестве последнего параметра можно указывать те же константы, что рассматривались для функции `WSAAsyncSelect` (все они начинаются с префикса `FD_`).

Раньше вы уже встречались с функциями `WaitForSingleObject` и `WaitForMultipleObjects`, которые ожидают наступления события типа `HANDLE`. Для сетевых событий используется похожая функция с именем `WSAWaitForMultipleEvents`:

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT FAR *lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeOUT,
    BOOL fAlertable
   );
```

Давайте рассмотрим каждый параметр:

- `cEvents` — количество объектов событий, изменение состояния которых нужно ожидать. Чтобы узнать максимальное число, воспользуйтесь константой `WSA_MAXIMUM_WAIT_EVENTS`;
- `lphEvents` — массив объектов событий, которые нужно ожидать;

- fWaitAll — режим ожидания событий. Если он TRUE, то функция ожидает, пока все события не сработают, иначе после первого передает управление программе;
- dwTimeOUT — временной интервал в миллисекундах, в течение которого нужно ожидать события. Если в этом временном интервале не возникло события, то функция возвращает значение WSA_WAIT_TIMEOUT. Если нужно ожидать бесконечно, то можно указать константу WSA_INFINITE;
- fAlertable — параметр используется при перекрестном вводе/выводе, который я не рассматриваю в этой книге, поэтому указан FALSE.

Чтобы узнать, какое событие из массива событий сработало, нужно вычесть из возвращенного функцией WSAWaitForMultipleEvents значения константу WSA_WAIT_EVENT_0.

Прежде чем вызывать функцию WSAWaitForMultipleEvents, все события в массиве должны быть пустыми. Если хотя бы одно из них будет занято, то функция сразу вернет управление программе, и не будет ожидания. Отработавшие события становятся занятыми, и после обработки их надо освободить. Для этого используется функция WSAResetEvent:

```
BOOL WSAResetEvent(
    WSAEVENT hEvent
);
```

Функция очищает состояние события, указанного в качестве единственного параметра.

Когда событие уже не нужно, его необходимо закрыть. Для этого используется функция WSACloseEvent. Функции следует передать объект события, который необходимо закрыть:

```
BOOL WSACloseEvent(
    WSAEVENT hEvent
    );
```

Если закрытие прошло успешно, то функция возвращает TRUE, иначе — FALSE.

ГЛАВА 5



Работа с железом

В этой главе я затрону вопросы, касающиеся аппаратной части компьютера. Для хакера очень важно знать, как работать с компьютерным железом и уметь определять его параметры.

Так как в данной книге упор сделан на работу с сетью, то и здесь я коснусь этой темы. Я покажу, как определить параметры сетевой карты и сетевые настройки. Прочитав эту главу, вы найдете ответ на вопрос, как узнать IP-адрес локального компьютера. Мы узнаем не только один адрес, а все параметры сетевой конфигурации.

Помимо этого достаточно подробно будет описана работа с COM-портами, которые используются при подключении к компьютеру различного оборудования. Когда я занимался автоматизацией производства, мне пришлось написать множество программ, которые через COM-порты собирают данные с устройств или наблюдают за работой аппаратуры.

5.1. Параметры сети

В Windows 9*x* была очень удобная и полезная утилита WinIPConfig, которая отображала параметры сети. С помощью этой утилиты легко можно было узнать IP-адрес каждого сетевого устройства или MAC-адрес.

Сетевой MAC-адрес является уникальным и прошит в памяти сетевого устройства. Это свойство MAC-адреса стали использовать для обеспечения безопасности или защиты программ. Если в компьютере есть сетевая карта, то ее уникальный номер получить достаточно просто.

Для работы с параметрами сети используется библиотека IPHlpApi.lib. Давайте рассмотрим пример, и на его основе я познакомлю вас с самыми интересными функциями этой библиотеки.

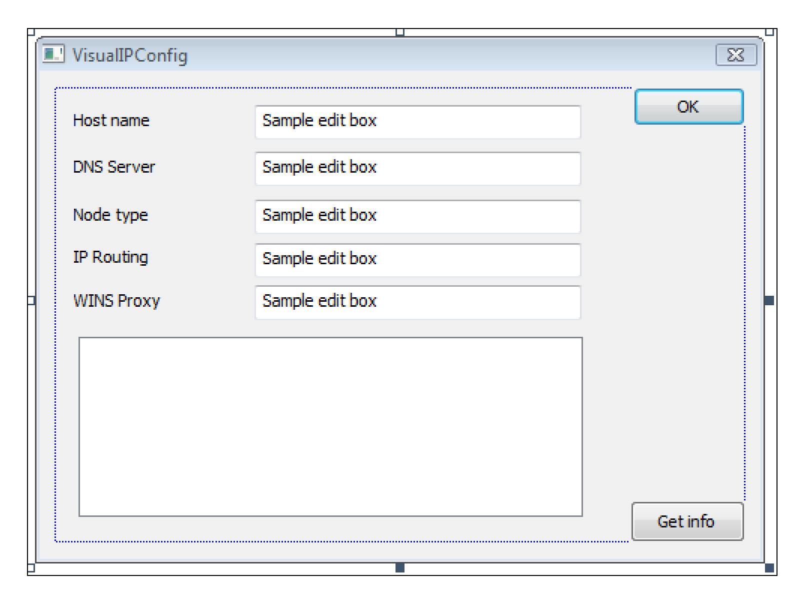


Рис. 5.1. Окно будущей программы VisualIPConfig

Создайте новое приложение MFC-Application на базе диалогового окна. Расположите в главном окне пять полей `Edit Control`, один `List Box` и кнопку с надписью "Get info". Окно, которое получилось у меня, вы можете увидеть на рис. 5.1.

Для полей ввода создайте следующие переменные: `eHostName`, `DNSServers`, `eNodeType`, `eIPRouting`, `eWinsProxy`. Для списка введите переменную `AdaptersInfo`.

Сетевых устройств может быть несколько, поэтому информация о них будет выводиться в список, а общая информация будет отображаться в полях ввода.

Создайте обработчик события `BN_CLICKED` для кнопки (для этого можно просто дважды щелкнуть по ней) и в него добавьте содержимое листинга 5.1. Я советую набрать код вручную, а не использовать пример с диска.

Листинг 5.1. Определение параметров сетевой карты

```
void CVisualIPConfigDlg::OnBnClickedButton1()
{
    PFIXED_INFO pFixedInfo;
    ULONG iFixedInfo = 0;

    PIP_ADAPTER_INFO pAdapterInfo, pAdapter;
    ULONG iAdapterInfo;
    PIP_ADDR_STRING chAddr;
```

```
CString Str;
TCHAR lpszText[1024];
int iErr;

if ((iErr = GetNetworkParams(NULL, &iFixedInfo)) != 0)
{
    if (iErr != ERROR_BUFFER_OVERFLOW)
    {
        AfxMessageBox("GetNetworkParams failed");
        return;
    }
}

if ((pFixedInfo=(PFIXED_INFO)GlobalAlloc(GPTR, iFixedInfo))!=NULL)
{
    AfxMessageBox("Memory allocation error");
    return;
}

if (GetNetworkParams(pFixedInfo, &iFixedInfo) != 0)
   {
    AfxMessageBox("GetNetworkParams failed");
    return;
}

eHostName.SetWindowText(pFixedInfo->HostName);

CString s = pFixedInfo->DnsServerList.IpAddress.String;
chAddr = pFixedInfo->DnsServerList.Next;
while(chAddr)
{
    s = s+" "+chAddr->IpAddress.String;
    chAddr = chAddr->Next;
}
DNSServers.SetWindowText(s);

switch (pFixedInfo->NodeType)
{
    case 1:
        eNodeType.SetWindowText("Broadcast");
        break;
    case 2:
        eNodeType.SetWindowText("Peer to peer");
        break;
```

```
        case 4:
            eNodeType.SetWindowText("Mixed");
            break;
        case 8:
            eNodeType.SetWindowText("Hybrid");
            break;
        default:
            eNodeType.SetWindowText("Don't know");
    }

    eIPRouting.SetWindowText(pFixedInfo->EnableRouting ? "Enabled" : "Disabled");
    eWinsProxy.SetWindowText(pFixedInfo->EnableProxy ? "Enabled" : "Disabled");

    iAdapterInfo = 0;
    iErr = GetAdaptersInfo(NULL, &iAdapterInfo);
    if ((iErr != 0) && (iErr != ERROR_BUFFER_OVERFLOW))
    {
              AfxMessageBox("GetAdaptersInfo failed");
              return;
    }

    if ((pAdapterInfo =
          (PIP_ADAPTER_INFO) GlobalAlloc(GPTR, iAdapterInfo)) == NULL)
    {
        AfxMessageBox("Memory allocation error\n");
        return;
    }

    if (GetAdaptersInfo(pAdapterInfo, &iAdapterInfo) != 0)
    {
        AfxMessageBox("GetAdaptersInfo failed");
        return;
    }

    pAdapter = pAdapterInfo;

    eAdaptersInfo.AddString("===========================");

    while (pAdapter)
    {
        switch (pAdapter->Type)
        {
            case MIB_IF_TYPE_ETHERNET:
                     Str = "Ethernet adapter: "; break;
```

```
            case MIB_IF_TYPE_PPP:
                     Str = "PPP adapter: "; break;
            case MIB_IF_TYPE_LOOPBACK:
                     Str = "Loopback adapter: "; break;
            case MIB_IF_TYPE_TOKENRING:
                     Str = "Token Ring adapter: "; break;
            case MIB_IF_TYPE_FDDI:
                     Str = "FDDI adapter: "; break;
            case MIB_IF_TYPE_SLIP:
                     Str = "Slip adapter: "; break;
            case MIB_IF_TYPE_OTHER:
            default: Str = "Other adapter: ";
        }
        eAdaptersInfo.AddString(Str+pAdapter->AdapterName);

        Str = "Description: ";
        eAdaptersInfo.AddString(Str+pAdapter->Description);

        Str = "Physical Address: ";
        for (UINT i = 0; i < pAdapter->AddressLength; i++)
        {
            if (i == (pAdapter->AddressLength - 1))
                sprintf(lpszText, "%.2X",(int)pAdapter->Address[i]);
            else
                sprintf(lpszText, "%.2X",(int)pAdapter->Address[i]);
            Str=Str+lpszText;
        }
        eAdaptersInfo.AddString(Str);

        sprintf(lpszText, "DHCP Enabled: %s",
                   (pAdapter->DhcpEnabled ? "yes" : "no"));
        eAdaptersInfo.AddString(lpszText);

        chAddr = &(pAdapter->IpAddressList);
        while(chAddr)
        {
             Str = "IP Address: ";
             eAdaptersInfo.AddString(Str+chAddr->IpAddress.String);

             Str="Subnet Mask: ";
             eAdaptersInfo.AddString(Str+chAddr->IpMask.String);

             chAddr = chAddr->Next;
        }
```

```
        Str = "Default Gateway: ";
        eAdaptersInfo.AddString(Str+pAdapter->GatewayList.IpAddress.String);

        chAddr = pAdapter->GatewayList.Next;
        while(chAddr)
        {
            // print next Gateway (Печатать следующий шлюз)
            chAddr = chAddr->Next;
        }

        Str = "DHCP Server: ";
        eAdaptersInfo.AddString(Str+pAdapter->DhcpServer.IpAddress.String);

        Str = "Primary WINS Server: ";
        eAdaptersInfo.AddString(Str+pAdapter->PrimaryWinsServer.IpAddress.String);

        Str = "Secondary WINS Server: ";
        eAdaptersInfo.AddString(Str+pAdapter->
        SecondaryWinsServer.IpAddress.String);

        eAdaptersInfo.AddString("===========================");
        pAdapter = pAdapter->Next;
    }
}
```

Общую информацию о сети можно получить с помощью функции `GetNetworkParams`. У нее два параметра: структура типа `PFIXED_INFO` и размер структуры.

Если первый параметр оставить нулевым, а в качестве второго указать числовую переменную, то в эту переменную запишется размер памяти, необходимый для структуры `PFIXED_INFO`. Именно это и делается в первый раз. Память надо выделять в глобальной области с помощью функции `GlobalAlloc`, иначе функция может вернуть неверные данные.

После этого функция `GetNetworkParams` вызывается еще раз, но с указанием двух параметров. Если результатом выполнения функции будет `0`, то получение данных прошло успешно.

Теперь разберем параметры структуры `PFIXED_INFO`:

- `HostName` — имя компьютера;
- `DnsServerList.IpAddress` — список IP-адресов серверов DNS;
- `NodeType` — тип сетевого устройства;

- EnableRouting — если равно TRUE, то маршрутизация включена;
- EnableProxy — если TRUE, то кэширование включено.

Получив общую информацию, можно приступить к перечислению параметров всех установленных адаптеров. Для этого используется функция GetAdaptersInfo. У нее также два параметра: переменная типа PIP_ADAPTER_INFO и размер. Если первый параметр нулевой, то через второй параметр функция вернет необходимый размер для структуры PIP_ADAPTER_INFO.

Рассмотрим параметры полученной структуры PIP_ADAPTER_INFO:

- Type — тип адаптера. Может принимать одно из следующих значений:
  - MIB_IF_TYPE_ETHERNET — сетевой адаптер Ethernet;
  - MIB_IF_TYPE_TOKENRING — адаптер Token Ring;
  - MIB_IF_TYPE_FDDI — адаптер FDDI;
  - MIB_IF_TYPE_PPP — PPP-адаптер;
  - MIB_IF_TYPE_LOOPBACK — адаптер LoopBack;
  - MIB_IF_TYPE_SLIP — Slip-адаптер;
  - MIB_IF_TYPE_OTHER — другое;
- AdapterName — имя адаптера;
- Description — описание, которое может хранить название фирмы-производителя или предназначение;
- AddressLength — длина MAC-адреса;
- Address — MAC-адрес;
- DhcpEnabled — принимает значение TRUE, если включен DHCP;
- IpAddressList — список IP-адресов и масок сети. Каждый адаптер может иметь одновременно несколько адресов;
- GatewayList — список шлюзов;
- DhcpServer — адреса DHCP-серверов;
- PrimaryWinsServer — адрес первичного WINS-сервера;
- SecondaryWinsServer — адрес вторичного WINS-сервера.

Для компиляции примера необходимо открыть свойства проекта и в разделе **Lnker/Input** добавить библиотеку IPHlpApi.lib в свойство **Additional Dependencies**. А в начале модуля нужно добавить описание заголовочного файла iphlpapi.h.

Запустите файл VisualIPConfig.exe. Нажмите кнопку **Get info**. Все сведения о сетевой карте, полученные с помощью программы, представлены на рис. 5.2.

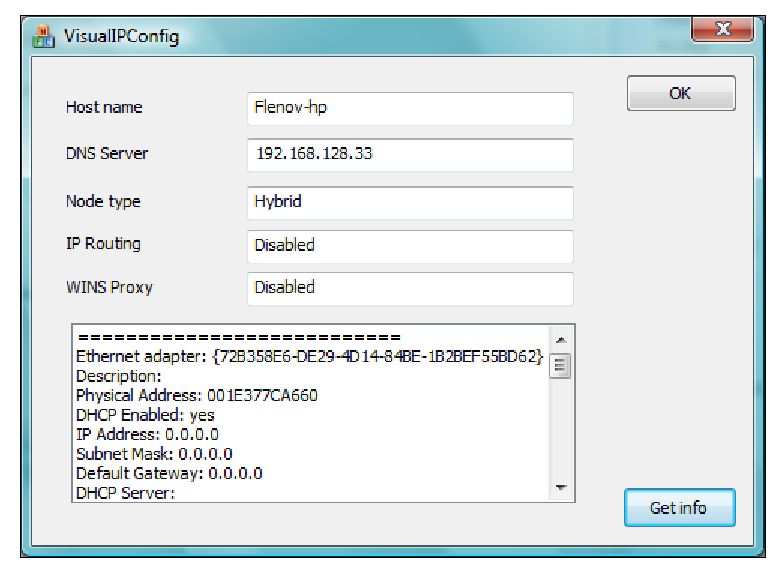


Рис. 5.2. Результат работы программы VisualIPConfig



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter5\VisualIPConfig.

5.2. Изменение IP-адреса

Попытаюсь ответить на часто задаваемый вопрос и объяснить, как программно поменять IP-адрес. Зная, как это делается, легко можно написать программу, которая будет через определенные промежутки времени менять адрес компьютера. Зачем это нужно? Таким образом можно создать своеобразный вид защиты. Когда компьютер меняет адрес, то он как бы появляется в новом месте, где хакер его не ожидает видеть, и хакеру трудно будет догадаться, что он имеет дело все с тем же компьютером. Правда, текущие соединения разорвутся, поэтому такой трюк можно проводить только с клиентскими машинами, ведь сервер должен быть всегда доступным.

Сетевая карта может иметь несколько адресов одновременно, поэтому есть функции для добавления и удаления адресов.

Для добавления воспользуйтесь функцией `AddIPAddress`, у которой следующие параметры:

- ❒ IP-адрес;
- ❒ маска сети для адреса;
- ❒ индекс адаптера, для которого добавляется адрес;
- ❒ контекст адреса. По своему опыту советую указывать нулевое значение. Контекст будет устанавливаться системой;
- ❒ экземпляр, который чаще всего оставляют нулевым.

Каждый IP-адрес привязывается к определенному адаптеру. Например, если в системе две сетевые карты, то для них будут формироваться две отдельные записи для хранения адреса. Контекст однозначно идентифицирует запись об адресе в системе. Не может быть двух записей с одним и тем же контекстом для одного или разных сетевых адаптеров.

Зная контекст адреса, вы можете легко удалить адрес с помощью функции `DeleteIPAddress`, которой нужно передать в качестве единственного параметра именно этот контекст.

Продемонстрирую все сказанное на примере. Для этого создайте новое приложение MFC Application на основе диалога с именем ChangeIPAddress. На рис. 5.3 показано главное окно будущей программы.

По нажатию кнопки **List adapters** информация о сетевых адресах будет получена и выведена в элементе `List Box`, который растянут вдоль нижней части окна. Код, который должен выполняться по нажатию этой кнопки, приведен в листинге 5.2.

Листинг 5.2. Вывод информации об установленных адресах

```
void CChangeIPAddressDlg::OnBnClickedButton3()
{
    PIP_ADAPTER_INFO pAdapterInfo, pAdapter;
    ULONG iAdapterInfo;
    int iErr;
    CString Str;

    iAdapterInfo = 0;
    iErr=GetAdaptersInfo(NULL, &iAdapterInfo);

    if ((iErr!= 0) && (iErr != ERROR_BUFFER_OVERFLOW))
    {
        AfxMessageBox("GetAdaptersInfo failed");
```

```
        return;
    }

    if ((pAdapterInfo = (PIP_ADAPTER_INFO) GlobalAlloc(GPTR,
              iAdapterInfo)) == NULL)
    {
        AfxMessageBox("Memory allocation error\n");
        return;
    }

    if (GetAdaptersInfo(pAdapterInfo, &iAdapterInfo) != 0)
    {
        AfxMessageBox("GetAdaptersInfo failed");
        return;
    }

    pAdapter = pAdapterInfo;
    lAdapters.AddString("======================");
       while (pAdapter)
       {
              Str="Adapter: ";
              lAdapters.AddString(Str+pAdapter->AdapterName);

              char s[20];
              Str=itoa(pAdapter->Index, s, 10);
              Str="Index: "+Str;
              lAdapters.AddString(Str);

              PIP_ADDR_STRING chAddr = &(pAdapter->IpAddressList);
              while(chAddr)
        {
              lAdapters.AddString("------------------------------");

              Str=itoa(chAddr->Context, s, 10);
              Str="Context: "+Str;
              lAdapters.AddString(Str);

              Str="IP Address: ";
              lAdapters.AddString(Str+chAddr->IpAddress.String);

              Str="Subnet Mask: ";
              lAdapters.AddString(Str+chAddr->IpMask.String);

              chAddr = chAddr->Next;
        }
```

```
        pAdapter = pAdapter->Next;
    }
}
```

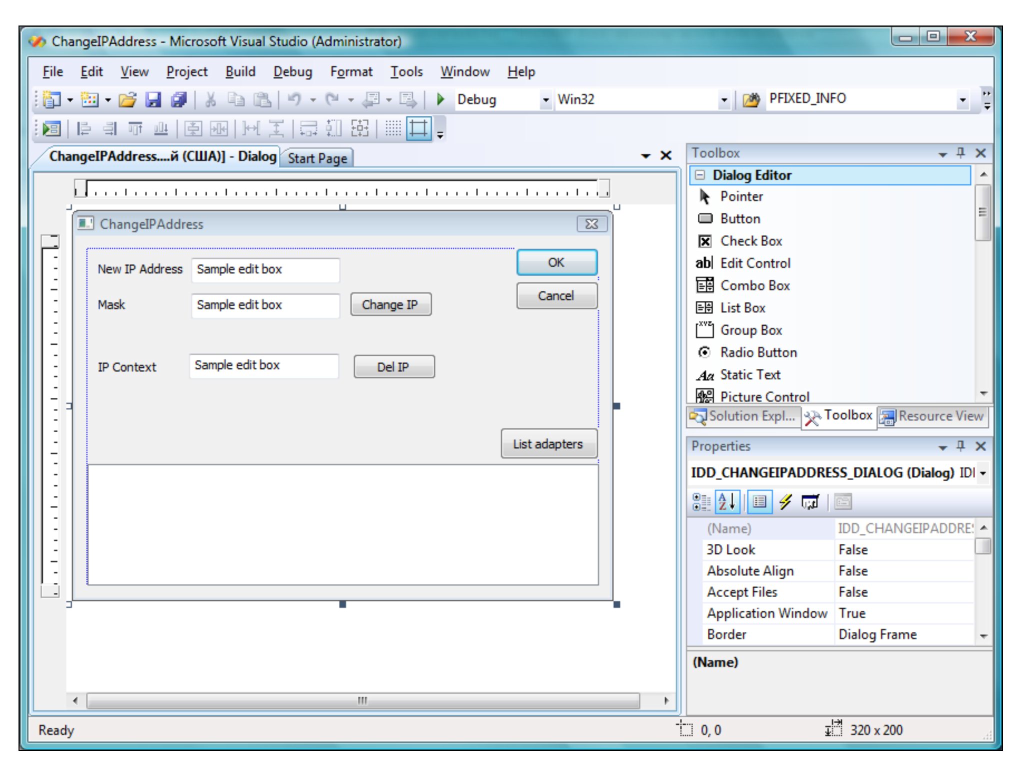


Рис. 5.3. Окно будущей программы ChangeIPAddress

Вся информация об адресах получена так же, как и в *разд. 5.1*, с помощью функции `GetAdaptersInfo`. Напомню, что в качестве первого параметра нужно указать структуру типа `PIP_ADAPTER_INFO`. В этой структуре в параметре `Index` хранится индекс сетевого устройства, который надо будет указывать в качестве третьего параметра функции `AddIPAddress` при добавлении нового IP, а в параметре `IpAddressList` — массив из структур типа `PIP_ADDR_STRING`. В этой структуре нас интересует параметр `Context`, в котором хранится контекст IP-адреса. В параметре `IpAddress` хранится адрес, а в `IpMask` находится маска сети.

По нажатию кнопки **Change IP** добавляется новый адрес для сетевого адаптера. Можно перед добавлением найти и удалить все уже существующие адреса, а потом присоединить новый. Пример кода, который должен выполняться по нажатию этой кнопки, приведен в листинге 5.3.

Листинг 5.3. Добавление нового адреса для первого сетевого адаптера в системе

```
void CChangeIPAddressDlg::OnBnClickedButton1()
{
    PIP_ADAPTER_INFO pAdapterInfo, pAdapter;
    ULONG iAdapterInfo;
    int iErr;
    ULONG iInst, iContext;
    iInst=iContext=0;

    iAdapterInfo = 0;
    iErr=GetAdaptersInfo(NULL, &iAdapterInfo);
    if ((iErr!= 0) && (iErr != ERROR_BUFFER_OVERFLOW))
    {
        AfxMessageBox("GetAdaptersInfo failed");
        return;
    }

    if ((pAdapterInfo = (PIP_ADAPTER_INFO) GlobalAlloc(GPTR,
                         iAdapterInfo)) == NULL)
    {
        AfxMessageBox("Memory allocation error\n");
        return;
    }

    if (GetAdaptersInfo(pAdapterInfo, &iAdapterInfo) != 0)
    {
        AfxMessageBox("GetAdaptersInfo failed");
        return;
    }

    pAdapter = pAdapterInfo;

    char sNewAddr[20], sNewMask[20];

    eIPEdit.GetWindowText(sNewAddr, 20);
    eMaskEdit.GetWindowText(sNewMask, 20);

    iErr=AddIPAddress(inet_addr(sNewAddr), inet_addr(sNewMask),
                         pAdapter->Index, &iContext, &iInst);
    if (iErr!=0)
           AfxMessageBox("Can't change address");
}
```

Чтобы добавить новый адрес, необходимо знать индекс сетевого адаптера. Для его определения используется функция `GetAdaptersInfo`. После этого можно вызывать функцию `AddIPAddress`.

По нажатию кнопки **Del IP** будет удаляться адрес с контекстом, указанным в поле **IP Context**. Код, который должен выполняться по нажатию этой кнопки, можно увидеть в листинге 5.4.

Листинг 5.4. Удаление IP-адреса

```
void CChangeIPAddressDlg::OnBnClickedButton2()
{
        char sContext[20];
        eContext.GetWindowText(sContext, 20);

        int Context=atoi(sContext);
        if (DeleteIPAddress(Context) != 0)
        {
                AfxMessageBox("Can't delete address");
        }
}
```

Интересного эффекта можно добиться, если просто удалить все IP-адреса. В этом случае компьютер исчезнет из сети и не сможет с ней работать. Но это уже из серии программ-шуток. В этой главе мы уже двинулись дальше.

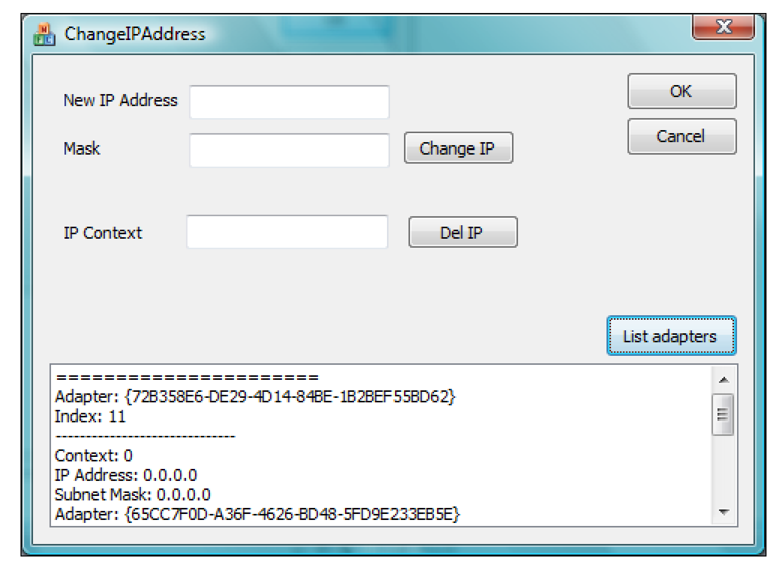


Рис. 5.4. Результат работы программы ChangeIPAddress без сетевого кабеля

Напоследок хотел бы вас предупредить, что функции будут корректно работать только при правильно настроенной сети. Даже если просто выдернуть сетевой кабель, функции не работают. На рис. 5.4 показан результат работы программы на моем ноутбуке. Перед нажатием кнопки **List adapters** я отключил сетевой кабель, и в результате IP-адрес и маска сети стали нулевыми (0.0.0.0).



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter5\ChangeIPAddress.

5.3. Работа с COM-портом

Мне по долгу службы часто приходилось работать с интерфейсом RS-232. Так в официальной документации называется COM-порт компьютера. Производственное оборудование (контроллеры, устройства сбора информации и т. д.) очень часто управляется через этот порт. К любому модему, даже внутреннему, обращение происходит именно через COM-порт. А сколько существует внешних устройств, подключаемых по этому интерфейсу, сосчитать невозможно.

Работа с портами похожа на работу с файлами. Давайте рассмотрим простейший пример. Для этого создайте новое приложение MFC Application на основе диалога с именем COMport. Размер для данного примера не имеет значения, поэтому я выбрал простоту и MFC. Внешний вид главного окна будущей программы вы можете увидеть на рис. 5.5.

В верхней части окна находится выпадающий список `ComboBox`, в котором можно выбирать имя порта. Рядом со списком две кнопки: для открытия и закрытия порта. Чуть ниже расположены текстовое поле для ввода команды и кнопка для ее отправки.

В центре окна расположились элементы управления `ListBox` для отображения хода работы с портом и многострочное поле ввода для отображения пришедших данных.

Создайте подобный интерфейс, и можно переходить к программированию. По нажатию кнопки **Open port** должен выполняться код из листинга 5.5.

Листинг 5.5. Открытие порта

```
void CCOMportDlg::OnBnClickedOpenportButton()
{
       if (hCom != INVALID_HANDLE_VALUE)
```

```
        {
                OnBnClickedButton1();
                Sleep(300);
        }

        char sPortName[10];
        cbPorts.GetWindowText(sPortName, 10);

        hCom = CreateFile(sPortName, GENERIC_READ | GENERIC_WRITE,
                                     0, NULL, OPEN_EXISTING, 0, NULL);

        if (hCom == INVALID_HANDLE_VALUE)
                lLogList.AddString("Error opening port");
        else
        {
                lLogList.AddString("Port successfully opened.");

                DCB dcb;
                GetCommState(hCom, &dcb);
                dcb.BaudRate = CBR_57600;
                dcb.ByteSize = 8;
                dcb.Parity = NOPARITY;
                dcb.StopBits = ONESTOPBIT;
                if (SetCommState(hCom, &dcb))
                        lLogList.AddString("Configuring OK");
                else
                        lLogList.AddString("Configuring Error");

                hThread = CreateThread(0, 0, ReadThread, (LPVOID)this,
                      0, 0);
        }
}
```

Если попытаться открыть порт дважды, то будет получено сообщение об ошибке, поэтому первым делом нужно произвести эту проверку. И если порт открыт, то закрыть его. Эта проверка выполняется в функции `OnBnClickedButton1`, которую я покажу чуть позже, и она будет вызываться при нажатии на кнопку **Close port**.

Теперь получим имя выбранного порта и откроем его. Для этого используется функция работы с простыми файлами `CreateFile`, только вместо имени файла указывается имя порта.

Если порт открыт удачно, то выводится соответствующее сообщение, и можно перейти к конфигурированию параметров соединения. Для этого сначала

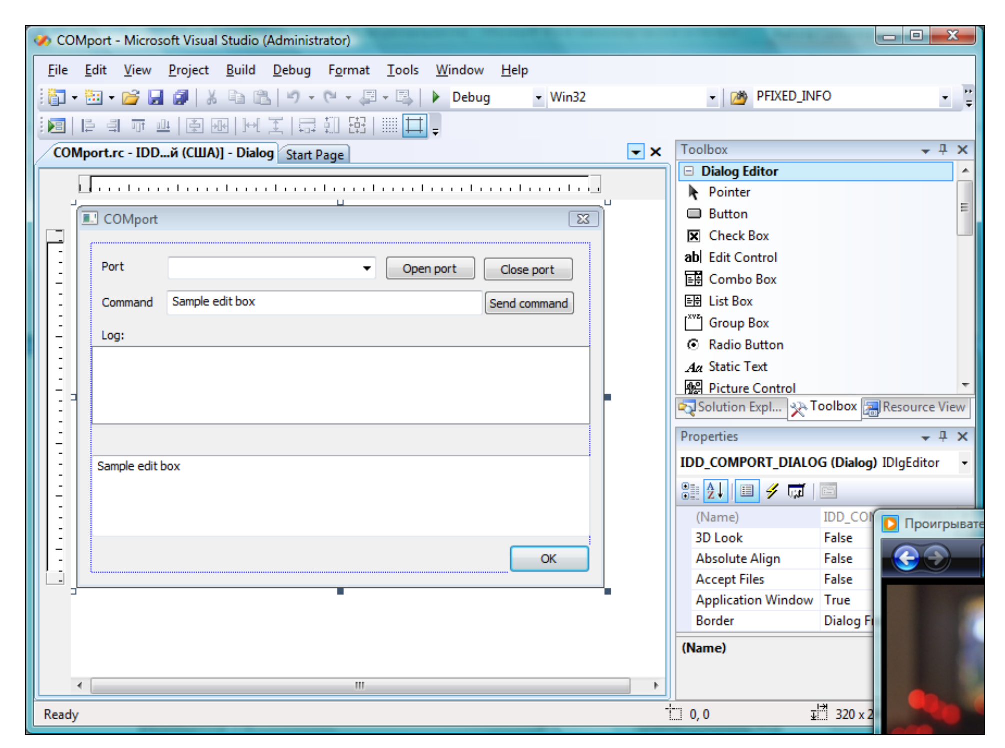


Рис. 5.5. Окно будущей программы Comport

получите текущие настройки системы с помощью функции `GetCommState`. Ей нужно передать два параметра: указатель на открытый порт и указатель на структуру типа `DCB`. Эта структура содержит полную информацию о параметрах соединения и выглядит следующим образом:

```
typedef struct _DCB {
    DWORD DCBlength;           // размер структуры DCB
    DWORD BaudRate;            // скорость передачи данных в бодах
    DWORD fBinary: 1;          // двоичный режим без проверки конца
                               // строки
    DWORD fParity: 1;          // включить проверку четности
    DWORD fOutxCtsFlow:1;      // CTS-управление потоком выхода
    DWORD fOutxDsrFlow:1;      // DSR-управление потоком выхода
    DWORD fDtrControl:2;       // DTR-тип управления потоком скорости
                               // передачи данных
    DWORD fDsrSensitivity:1;   // DSR-чувствительность

    DWORD fTXContinueOnXoff:1; // стоп-сигнал продолжает выполнение
    DWORD fOutX: 1;            // старт/стоп-сигнал для управления
                               // выходящим потоком
```

```
    DWORD fInX: 1;              // старт/стоп-сигнал для управления
                                // входящим потоком

    DWORD fErrorChar: 1;        // включить проверку погрешностей
    DWORD fNull: 1;             // отвергать пустой поток данных
    DWORD fRtsControl:2;        // RTS-управление потоком данных
    DWORD fAbortOnError:1;      // проверять операции чтения/записи
    DWORD fDummy2:17;           // зарезервировано
    WORD wReserved;             // зарезервировано
    WORD XonLim;                // порог чувствительности старт-сигнала
    WORD XoffLim;               // порог чувствительности стоп-сигнала

    BYTE ByteSize;              // количество битов (обычно 7 или 8)
    BYTE Parity;                // четность байта
    BYTE StopBits;              // стоповые биты
    char XonChar;               // вид старт-сигнала в потоке
    char XoffChar;              // вид стоп-сигнала в потоке
    char ErrorChar;             // вид сигнала погрешности

    char EofChar;               // сигнал окончания потока
    char EvtChar;               // зарезервировано
    WORD wReserved1;            // зарезервировано
} DCB;
```

Если неправильно указаны параметры, то данные не будут передаваться и приниматься. Самое главное — заполнить следующие поля:

- `BaudRate` — скорость передачи данных (бит/с). Указывается константа в виде `CBR_скорость`, где `скорость` должна быть равна скорости, поддерживаемой используемым устройством, например, 56 000;
- `ByteSize` — размер передаваемого байта (может быть 7 или 8);
- `Parity` — флаг проверки четности;
- `StopBits` — стоповые биты, могут принимать значения `ONESTOPBIT` (один), `ONE5STOPBITS` (полтора) или `TWOSTOPBITS` (два).

Остальные параметры можно оставить по умолчанию (те, которые вернула система). Но прежде чем указывать какие-либо параметры, обязательно прочитайте документацию на аппаратуру, с которой необходимо соединяться. Я не встречал устройств, которые поддерживали бы все режимы работы. Например, модем ZyXel Omni 56K может поддерживать скорость от 2400 до 56 000, и можно указывать значения только из этого диапазона.

Помимо этого нужно, чтобы оба устройства (передающее и принимающее) были настроены одинаково (скорость, размер байта и т. д.), иначе данные не будут передаваться.

После конфигурирования порта запускается поток, в котором мы будем бесконечно пытаться считать данные из порта. Это, конечно же, не эффективно, потому что удобнее использовать сообщения Windows, но для простого примера в обучающих целях достаточно. Функция чтения потока `ReadThread` выглядит следующим образом:

```
DWORD __stdcall ReadThread(LPVOID hwnd)
{
   DWORD iSize;
   char sReceivedChar;
   while(true)
   {
       ReadFile(hCom,&sReceivedChar,1,&iSize,0);
       SendDlgItemMessage((HWND)hwnd,IDC_EDIT2,WM_CHAR,
                sReceivedChar,0);
   }
}
```

В этой функции вы можете увидеть бесконечный цикл чтения данных, которое выполняется стандартной функцией чтения из файла — `ReadFile`.

Теперь посмотрите на функцию закрытия порта, которая будет вызываться по нажатию кнопки **Close port**:

```
void CCOMportDlg::OnBnClickedButton1()
{
       if (hCom == INVALID_HANDLE_VALUE)
              return;

       if (MessageBox("Close port?", "Warning", MB_YESNO) == IDYES)
       {
              TerminateThread(hThread, 0);
              CloseHandle(hCom);
              hCom = INVALID_HANDLE_VALUE;
       }
}
```

Прежде чем выполнить закрытие, надо проверить переменную `hCom`. Возможно, что порт уже закрыт или вообще никогда не открывался. Если переменная содержит неправильный указатель, то следует просто выйти из функции.

Если порт открыт, то выводится запрос на подтверждение закрытия порта. Если пользователь подтвердит, то прерывается поток, закрывается указатель порта и переменной `hCom` присваивается значение `INVALID_HANDLE_VALUE`.

И последнее, что предстоит добавить в программу — возможность отправки сообщений. Для этого по нажатию кнопки **Send command** должен выполняться код из листинга 5.6.

Листинг 5.6. Функция отправки данных в порт

```
void CCOMportDlg::OnBnClickedSendcommandButton()
{
       if (hCom == INVALID_HANDLE_VALUE)
       {
              AfxMessageBox("Open port before send command");
              return;
       }

       char sSend[10224];
       eSendCommand.GetWindowText(sSend, 1024);

       if (strlen(sSend)>0)
       {
            lLogList.AddString(sSend);

            sSend[strlen(sSend)] = '\r';
            sSend[strlen(sSend)] = '\0';

            TerminateThread(hThread,0);
            DWORD iSize;
            WriteFile(hCom, sSend, strlen(sSend), &iSize,0);
            hThread = CreateThread(0, 0, ReadThread,
                              (LPVOID)this, 0, 0);
       }
}
```

Сначала проверяется, открыт ли порт. Если он уже закрыт или никогда не открывался, то нет смысла писать в него данные. После этого надо получить данные для отправки, и если они больше нуля, то добавить в конец отправляемой строки символ завершения строки (нулевой символ). Мне приходилось работать с разным оборудованием, и большинство типов требует в конце команды отправлять символы конца строки и перевода каретки. Иногда бывает достаточно только символа конца строки.

Теперь прерываем поток чтения данных и записываем в порт данные стандартной функцией работы с файлами `WriteFile`. После записи можно снова запускать поток чтения.

Если у вас есть модем, то можете запустить программу и открыть порт, на котором настроен модем. Отправьте команду `ATDTxxxxx`, где `xxxx` — это номер телефона. Модем должен будет начать набор указанного номера телефона.



Исходный код примера, описанного в этом разделе, вы можете найти в каталоге \Demo\Chapter5\COMport.

5.4. Подвисшие файлы

Вы уже немного познакомились с файлами, а в *разд. 4.2* даже попробовали создать файл на удаленной машине и записать в него данные. Теперь посмотрим, как можно превратить работу с файлами в небольшую шалость.

Вспомните, как формируется сетевой путь:

```
\\Имя компьютера\диск\путь
```

Если нужно обратиться к локальному диску как сетевому, и при этом диск не является открытым, то после имени диска необходимо поставить знак `$`. Например, чтобы получить доступ к файлу myfile.txt на диске C:, нужно написать следующий путь:

```
\\MyComputer\C$\myfile.txt
```

Теперь самое интересное. В Windows нельзя создавать файлы с именами, содержащими знак вопроса, и проверка на такой знак делается на уровне ОС. Но если обращаться к файлу по сети, то проверка не выполняется. Если у вас есть сеть, то подключите любой сетевой диск другого компьютера. Допустим, что этому диску будет назначена буква "e". Теперь, если выполнить код из листинга 5.7, то программа зависнет, и снять ее будет невозможно.

Листинг 5.7. Создание неправильного файла по сети

```
if ((FileHandle = CreateFile("\\\\notebook\\e$\\?myfile.txt",
              GENERIC_WRITE | GENERIC_READ,
              FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
              CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
              )) == INVALID_HANDLE_VALUE)
{
       MessageBox(0, "Create file error", "Error",0);
       return;
}

// Write to file 9 symbols
if (WriteFile(FileHandle, "Test line", 9, &BWritten, NULL)== 0)
```

```
{
       MessageBox(0, "Write to file error", "Error",0);
       return;
}

// Close file
CloseHandle(FileHandle);
```

В листинге 5.7 я пытался создать файл и записать в него 9 символов, как и в *разд. 4.2*. Но имя файла написано неверно (присутствует символ `?`), поэтому создание невозможно, а проверка на недопустимый символ отсутствует. Именно поэтому программа зависает в ожидании ответа от ОС, которого не будет.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter5\TestFile.

ГЛАВА 6



Полезные примеры

В книге было приведено много шуточных задач и исследованы некоторые теоретические аспекты сетевого программирования. Теперь я продемонстрирую на примерах кое-какие полезные алгоритмы. С их помощью вы узнаете еще много любопытного о приемах хакеров, и заодно закрепим полученные теоретические знания.

При разборе сетевых функций в *главах 4* и *5* были рассмотрены интересные примеры, но в них было несколько недостатков. Например, сканер портов был медленным, и на проверку 1000 портов уходило слишком много времени *(см. разд. 4.4)*. Я уже упоминал о том, что необходимо для ускорения этого процесса. В этой главе я покажу быстрый сканер, который можно сделать очень гибким и универсальным.

Помимо этого я покажу, как улучшить прием/передачу данных. Именно эта часть чаще всего является узким местом в обеспечении максимальной производительности при минимальной нагрузке на процессор.

В программировании очень много нюансов, и в разных ситуациях для достижения максимального эффекта можно поступить по-разному. Рассмотреть абсолютно все я не смогу, потому что на это понадобятся тысячи страниц и потребуются глубокие знания математики, поэтому затрону в основном сетевую часть.

6.1. Алгоритм приема/передачи данных

В *разд. 4.10.2* мы уже рассмотрели пример, в котором сервер асинхронно ожидает соединения с помощью функции `select`, и как только происходило подключение, создавался новый поток, который обменивался данными с клиентом. Самое узкое место в этом примере — второй поток, который работал только с одним клиентом.

Я уже говорил о том, что с помощью асинхронной работы сетевых функций можно легко реализовать возможность работы сразу с несколькими клиентами. Да и отдельный поток для обмена сообщениями в данном случае является излишним. В листинге 6.1 приведен пример, в котором сервер ожидает соединения и работает с клиентом в одной функции, но может обслуживать сразу несколько клиентов.

Листинг 6.1. Алгоритм асинхронной работы с клиентом

```
DWORD WINAPI NetThread(LPVOID lpParam)
{
   SOCKET sServerListen;
   SOCKET ClientSockets[50];
   int TotalSocket=0;

   struct sockaddr_in localaddr, clientaddr;
   int iSize;

   sServerListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
   if (sServerListen == SOCKET_ERROR)
   {
       MessageBox(0, "Can't load WinSock", "Error", 0);
       return 0;
   }

   ULONG ulBlock;
   ulBlock = 1;
   if (ioctlsocket(sServerListen, FIONBIO, &ulBlock) == SOCKET_ERROR)
       return 0;

   localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   localaddr.sin_family = AF_INET;
   localaddr.sin_port = htons(5050);

   if (bind(sServerListen, (struct sockaddr *)&localaddr,
           sizeof(localaddr)) == SOCKET_ERROR)
   {
       MessageBox(0, "Can't bind", "Error", 0);
       return 1;
   }

   MessageBox(0, "Bind OK", "Error", 0);

   listen(sServerListen, 4);
```

```
MessageBox(0, "Listen OK", "Error", 0);

FD_SET ReadSet;
int ReadySock;

while (1)
{
     FD_ZERO(&ReadSet);
     FD_SET(sServerListen, &ReadSet);

     for (int i=0; i<TotalSocket; i++)
         if (ClientSockets[i] != INVALID_SOCKET)
              FD_SET(ClientSockets[i], &ReadSet);


     if ((ReadySock = select(0, &ReadSet, NULL, NULL, NULL)) ==
                SOCKET_ERROR)
     {
          MessageBox(0, "Select filed", "Error", 0);
     }

     //We have new connection (Есть новые подключения)
     if (FD_ISSET(sServerListen, &ReadSet))
     {
          iSize = sizeof(clientaddr);
          ClientSockets[TotalSocket] = accept(sServerListen,
                         (struct sockaddr *)&clientaddr, &iSize);
          if (ClientSockets[TotalSocket] == INVALID_SOCKET)
          {
                MessageBox(0, "Accept filed", "Error", 0);
                break;
          }
          TotalSocket++;
     }

     //We have data from client (Есть данные от клиента)
     for (int i=0; i<TotalSocket; i++)
     {
          if (ClientSockets[i] == INVALID_SOCKET)
               continue;
          if (FD_ISSET(ClientSockets[i], &ReadSet))
          {
              char szRecvBuff[1024],
              szSendBuff[1024];
```

```
                int ret=recv(ClientSockets[i], szRecvBuff, 1024, 0);
                if (ret == 0)
                {
                    closesocket(ClientSockets[i]);
                    ClientSockets[i]=INVALID_SOCKET;
                    break;
                }
                else if (ret == SOCKET_ERROR)
                {
                    MessageBox(0, "Recive data filed", "Error", 0);
                    break;
                }
                szRecvBuff[ret] = '\0';

                strcpy(szSendBuff, "Command get OK");

                ret = send(ClientSockets[i], szSendBuff,
                               sizeof(szSendBuff), 0);
                if (ret == SOCKET_ERROR)
                      break;
            }
        }
    }
    closesocket(sServerListen);
    return 0;
}
```

Рассмотрим, как работает этот пример. Секрет заключается в том, что объявлено две переменные:

- `sServerListen` — переменная типа `SOCKET`, которая будет использоваться для прослушивания порта и ожидания соединения со стороны клиента;
- `ClientSockets` — массив из 50 элементов типа `ClientSockets`. Этот массив будет использоваться для работы с клиентами, и именно 50 клиентов смогут обслуживаться одновременно. В данном примере каждому соединению будет выделяться очередной сокет, поэтому после пятидесятого произойдет ошибка. В реальной программе этот массив необходимо сделать динамическим, чтобы при отключении клиента можно было удалять из массива соответствующий сокет. Но чтобы не мучаться с динамикой, можно пойти на такое небольшое ограничение, тем более — в обучающих целях.

После этого создается сокет `sServerListen` для прослушивания сервера, переводится в асинхронный режим, связывается функцией `bind` с локальным ад-

ресом, и запускается прослушивание. В этом участке кода никаких изменений не произошло.

Самое любопытное происходит в бесконечном цикле, который раньше просто ожидал соединения. Теперь в набор добавляется не только сокет сервера, но и активные клиентские сокеты. После этого функция `select` ожидает, когда какой-либо из этих сокетов будет готов к чтению данных.

Дальше — еще интереснее. Первым делом проверяется серверный сокет. Если он готов к чтению, то присоединился клиент. Соединение принимается с помощью функции `accept`, а результат (сокет для работы с клиентом) сохраняется в последнем (доступном) элементе массива `ClientSockets`. После этого функция `select` будет ожидать событий и от этого клиента.

На следующем этапе проверяются все сокеты из массива на готовность чтения данных с их стороны. Если какой-нибудь клиент готов, то читаются данные и отправляется ответ. Если при чтении данные не получены, и функция `recv` вернула нулевое значение, то клиент отключился от сервера.

Этот алгоритм достаточно быстрый и универсальный. А главное, позволяет с помощью одного цикла обрабатывать серверный и клиентские сокеты. Это очень удобно и эффективно. Если нужно обмениваться небольшими сообщениями, то программу можно использовать уже в таком виде. Если будет происходить обмен данными большого объема, то необходимо добавить возможность чтения и отправки всех пришедших данных.

Не забудьте только заменить массив клиентских сокетов на динамический. Если вы не хотите использовать динамические массивы, то можно поступить проще: перед каждым заполнением структуры `FD_SET` упорядочивать в ней элементы, чтобы убрать сокеты, равные `INVALID_SOCKET`. После этого необходимо установить переменную `TotalSocket` так, чтобы она указывала на следующий после последнего реально существующего элемента массива.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter6\AdvancedTCPServer.

6.2. Самый быстрый сканер портов

Потоки — это очень мощная и удобная вещь, позволяющая создать многозадачность даже внутри отдельного приложения. Но у них есть один большой недостаток: программисты, познакомившись с потоками, начинают использовать их везде, где это надо и не надо.

Я видел много сканеров, которые используют по 20–50 потоков для одновременного сканирования большого количества портов. Понимаю, что пример,

который мы рассмотрели в *главе 4*, был очень медленным, и надо ускорять, но не таким же методом! Сейчас вам предстоит увидеть, как можно реализовать быстрое сканирование портов без использования потоков. А тогда как? Конечно, с помощью асинхронной работы с сетью. Можно создать несколько асинхронных сокетов и запустить ожидание соединения. Потом собрать все сокеты в набор `fd_set` и выполнить функцию `select` в ожидании события соединения с сервером. После ее выполнения необходимо проверить все сокеты на удачное соединение и вывести результат.

Давайте попробуем реализовать это на примере. Для иллюстрации сказанного создайте новое приложение MFC Application на основе диалогового окна. При этом не включайте опцию поддержки WinSock в разделе **Advanced Features**. В данном случае мы будем использовать некоторые функции WinSock2. Поэтому подключите заголовочный файл winsock2.h вручную и укажите в свойствах проекта необходимость использования библиотеки ws2_32.lib.

Теперь откройте в редакторе ресурсов главное окно программы. Оформите его в соответствии с рис. 6.1. Здесь необходимо добавить три поля ввода `Edit Box`,

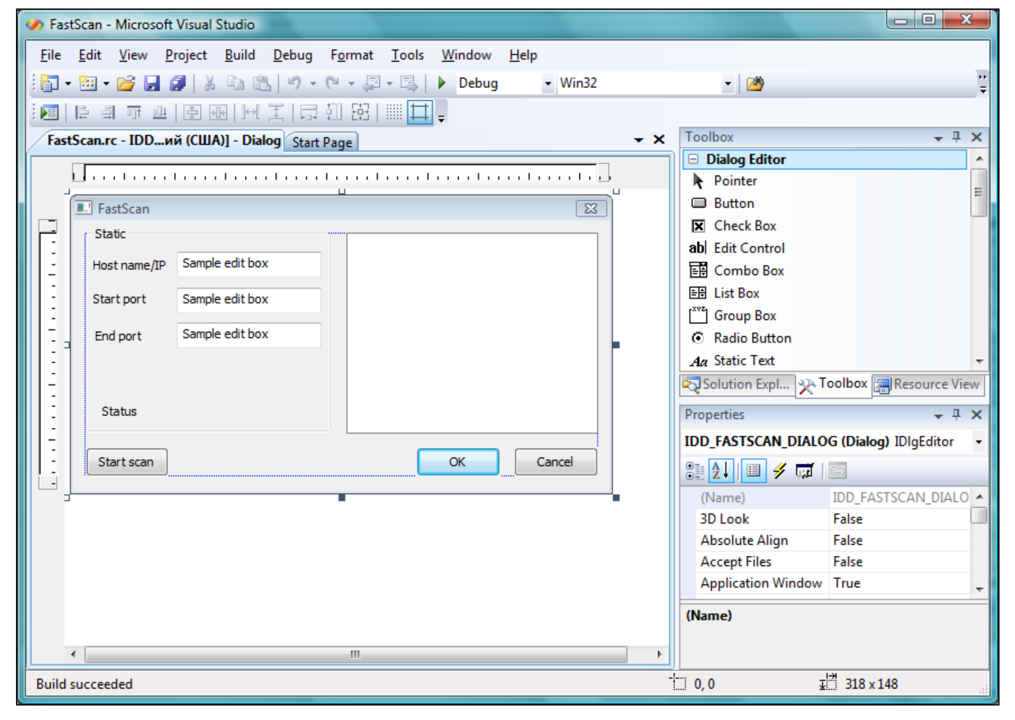


Рис. 6.1. Окно будущей программы FastScan

список List Box и кнопку, при нажатии которой будет происходить сканирование. Для всех полей ввода нужно создать следующие переменные:

- chHostName — имя или IP-адрес сканируемого компьютера;
- chStartPort — порт, с которого надо начать сканирование;
- chEndPort — порт, до которого нужно сканировать.

Портов очень много, и даже наш быстрый сканер потребует немало времени.

Теперь перейдем к программированию. Создайте обработчик события BN_CLICKED для кнопки запуска сканирования. Код, который здесь нужно написать, достаточно большой (листинг 6.2), но несмотря на то, что он есть на компакт-диске, я советую набрать его вручную. Только в этом случае вы сможете разобраться в предназначении каждой строчки. Я же постараюсь дать вам всю необходимую информацию.

Листинг 6.2. Быстрое сканирование портов

```
void CFastScanDlg::OnBnClickedButton1()
{
   char tStr[255];
   SOCKET sock[MAX_SOCKETS];
   int busy[MAX_SOCKETS], port[MAX_SOCKETS];
   int iStartPort, iEndPort, iBusySocks = 0;
   struct sockaddr_in addr;
   fd_set fdWaitSet;

   WSADATA wsd;
   if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
   {
      SetDlgItemText(IDC_STATUSTEXT, "Can't load WinSock");
      return;
   }

   SetDlgItemText(IDC_STATUSTEXT, "Resolving host");

   chStartPort.GetWindowText(tStr, 255);
   iStartPort = atoi(tStr);
   chEndPort.GetWindowText(tStr, 255);
   iEndPort = atoi(tStr);

   chHostName.GetWindowText(tStr, 255);

   struct hostent *host=NULL;
   host = gethostbyname(tStr);
```

```
if (host == NULL)
{
    SetDlgItemText(IDC_STATUSTEXT, "Unable to resolve host");
    return;
}

for (int i = 0; i < MAX_SOCKETS; i++)
    busy[i] = 0;

SetDlgItemText(IDC_STATUSTEXT, "Scanning");

while (((iBusySocks) || (iStartPort <= iEndPort)))
{
   for (int i = 0; i < MAX_SOCKETS; i++)
   {
       if (busy[i] == 0 && iStartPort <= iEndPort)
       {
           sock[i] = socket (AF_INET, SOCK_STREAM, PPROTO_TCP);
           if (sock[i] < 0)
           {
               SetDlgItemText(IDC_STATUSTEXT, "Socket filed");
               return;
           }
           iBusySocks++;
           addr.sin_family = AF_INET;
           addr.sin_port = htons (iStartPort);
           CopyMemory(&addr.sin_addr, host->h_addr_list[0],
                                     host->h_length);

          ULONG ulBlock;
          ulBlock = 1;
          if (ioctlsocket(sock[i], FIONBIO,
                                    &ulBlock) == SOCKET_ERROR)
          {
              return;
          }

          connect (sock[i], (struct sockaddr *)
                                       &addr, sizeof (addr));
           if (WSAGetLastError() == WSAEINPROGRESS)
           {
                closesocket (sock[i]);
                iBusySocks--;
           }
```

```
            else
            {
                 busy[i] = 1;
                 port[i] = iStartPort;
            }
            iStartPort++;
        }
    }
    FD_ZERO (&fdWaitSet);
    for (int i = 0; i < MAX_SOCKETS; i++)
    {
        if (busy[i] == 1)
             FD_SET (sock[i], &fdWaitSet);
    }

    struct timeval tv;
    tv.tv_sec = 1;
    tv.tv_usec = 0;
    if (select (1, NULL, &fdWaitSet, NULL, &tv) == SOCKET_ERROR)
    {
        SetDlgItemText(IDC_STATUSTEXT, "Select error");
        return;
    }

    for (int i = 0; i < MAX_SOCKETS; i++)
    {
        if (busy[i] == 1)
        {
            if (FD_ISSET (sock[i], &fdWaitSet))
            {
                int opt;
                int Len = sizeof(opt);
                if (getsockopt(sock[i], SOL_SOCKET, SO_ERROR,
                              (char*)&opt, &Len) == SOCKET_ERROR)
                     SetDlgItemText(IDC_STATUSTEXT, "error");

                if (opt == 0)
                {
                      struct servent *tec;
                      itoa(port[i],tStr, 10);
                      strcat(tStr, " (");
                      tec = getservbyport(htons(port[i]), "tcp");
                      if (tec==NULL)
                               strcat(tStr, "Unknown");
```

```
                        else
                                strcat(tStr, tec->s_name);

                        strcat(tStr, ") — open");
                        m_PortList.AddString(tStr);
                        busy[i] = 0;
                        shutdown(sock[i], SD_BOTH);
                        closesocket(sock[i]);
                    }
                    busy[i] = 0;
                    shutdown (sock[i], SD_BOTH);
                    closesocket (sock[i]);
                    iBusySocks--;
                }
                else
                {
                    busy[i] = 0;
                    closesocket(sock[i]);
                    iBusySocks--;
                }
            }
        }
    }
    WSACleanup();
    SetDlgItemText(IDC_STATUSTEXT, "Scaning complete");
    return;
}
```

В данном примере для сканирования используются три массива:

- `sock` — массив дескрипторов сокетов, которые ожидают соединения;
- `busy` — состояние сканируемых портов. Любой из них может быть занят и вызвать ошибку. В файле помощи по WinSock написано, что не каждый порт можно использовать. Поэтому элемент массива, номер которого соответствует такому занятому (зарезервированному) порту, делается равным `1`, в противном случае — присваивается `0`;
- `port` — массив сканируемых портов. В принципе можно было бы обойтись и без этого массива, но для упрощения кода я его ввел.

В этом примере есть одна новая функция, которую мы не рассматривали — `getservbyport`. Она выглядит следующим образом:

```
struct servent FAR * getservbyport (
    int port,
    const char FAR* proto
);
```

Функция возвращает информацию о сервисе порта, указанного первым параметром. Второй параметр определяет протокол. В качестве результата возвращается структура типа `servent`, в которой поле `s_name` содержит символьное описание сервиса. Если функция вернет нулевое значение, то невозможно определить по номеру порта параметры работающего сервиса.

Данные, которые возвращает функция `getservbyport`, не являются точными, и ее легко обмануть. Например, для порта с номером 21 функция будет всегда возвращать информацию о протоколе FTP (File Transfer Protocol), но никто вам не мешает запустить Web-сервер, и функция `getservbyport` не сможет этого определить. А точнее, она даже не будет пытаться.

Все остальное вам уже должно быть знакомо, но я подведу итоги, описав используемый алгоритм:

1. Загрузить сетевую библиотеку.
2. Определить адрес сканируемого компьютера до начала цикла. Этот адрес будет использоваться внутри цикла перебора портов в структуре `sockaddr_in`. Сама структура будет заполняться в цикле, потому что каждый раз будет новый порт, а адрес изменяться не будет, поэтому его определение вынесено за пределы цикла. Нет смысла на каждом этапе цикла делать одну и ту же операцию, тем более что определение IP-адреса может занять время, если указано имя сканируемого компьютера.
3. Запустить цикл, который будет выполняться, пока начальный порт не превзойдет конечный. Внутри этого большого цикла выполняются следующие действия:
   - запустить цикл от 0 до значения `MAX_SOCKETS`. В этом цикле создается сокет, переводится в асинхронный режим и запускается функция `connect`. Так как сокеты находятся в асинхронном режиме, то не будет происходить ожидания соединения и замораживания программы, но при этом и неизвестно, произошло соединение или нет;
   - обнулить переменную `fdWaitSet` типа `fd_set`;
   - запустить цикл от 0 до значения `MAX_SOCKETS`. В этом цикле все сокеты помещаются в набор `fd_set`;
   - ожидать события от сокета с помощью функции `select`;
   - запустить цикл от 0 до значения `MAX_SOCKETS`. В этом цикле проверяется, какие сокеты удачно соединились с сервером. Если соединение прошло успешно, то получить символьное имя порта с помощью функции `getsockopt`. После этого сокет закрыть, чтобы разорвать соединение с сервером.
4. Выгрузить сетевую библиотеку.

Что такое MAX_SOCKETS? Это константа, которая определяет количество сканируемых сокетов. В данном примере она равна 40, и это оптимальное значение для различных сред. Чем больше количество сокетов, сканируемых за один проход, тем быстрее все будет проходить, но слишком большое значение может привести к тому, что один из портов может не успеть ответить или удаленная система безопасности сработает и заблокирует вашу попытку сканирования.

Еще один недостаток: сканирование блокирует работу программы, поэтому открытые порты вы сможете увидеть только после окончания сканирования, когда программа освободится и перерисует окно. Чтобы избежать заморозки, можно написать следующую процедуру:

```
void ProcessMessages()
{
    MSG msg;
    while (PeekMessage(&msg,NULL,0,0,PM_NOREMOVE))
    {
        if (GetMessage(&msg, NULL, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            return;
    }
}
```

Эта функция содержит простой цикл — обработчик сообщений, который вы уже не раз видели в Win32-приложениях. В данном случае он не бесконечный и обрабатывает все сообщения, накопившиеся в очереди. А когда они заканчиваются, цикл прерывается, и программа будет продолжать сканирование.

Напишите саму функцию где-нибудь в начале модуля и вставьте вызов ProcessMessages() в конце цикла поиска портов. В этом случае вы избавитесь от заморозки и сможете увидеть открытые порты сразу.

Стоит еще заметить, что в данном случае использовался протокол, который отображает открытые TCP-порты. Он никак не связан с UDP-портами. Чтобы сканировать UDP, необходимо создавать сокет (функция socket), ориентированный на сообщения.

Вот и все, наш быстрый сканер портов готов. На рис. 6.2 вы можете увидеть пример результата выполнения программы.

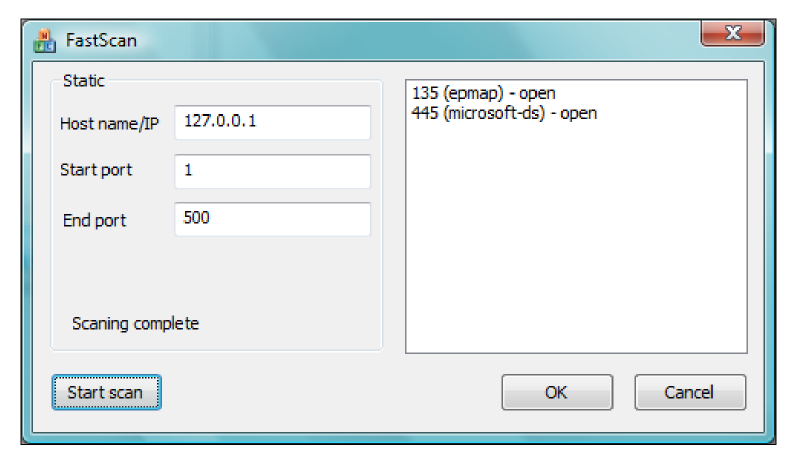


Рис. 6.2. Результат сканирования моего компьютера



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter6\FastScan.

6.3. Состояние локального компьютера

Если нужно узнать состояние портов локального компьютера, нет необходимости сканировать порты. Есть способ лучше — запросить состояние всех портов с помощью функции `GetTcpTable`. В этом случае вы получите более подробную информацию, которую можно свести в таблицу из следующих колонок:

- локальный адрес — интерфейс, на котором открыт порт;
- локальный порт — открытый порт;
- удаленный адрес — адрес, с которого в данный момент установлено соединение с портом;
- удаленный порт — порт на удаленной машине, через который происходит обращение к локальной машине;
- состояние — может принимать различные значения: прослушивание, закрытие порта, принятие соединения и т. д.

Самое главное преимущество использования состояния локальной таблицы TCP — мгновенная работа. Сколько бы ни было открытых портов, их определение происходит в считанные миллисекунды.

Для иллюстрации примера работы с TCP-портом создайте MFC-приложение с именем IPState на основе диалогового окна. На главное диалоговое окно

поместите один список типа `List Box` и кнопку с заголовком **TCP Table**. На рис. 6.3 вы можете увидеть окно будущей программы.

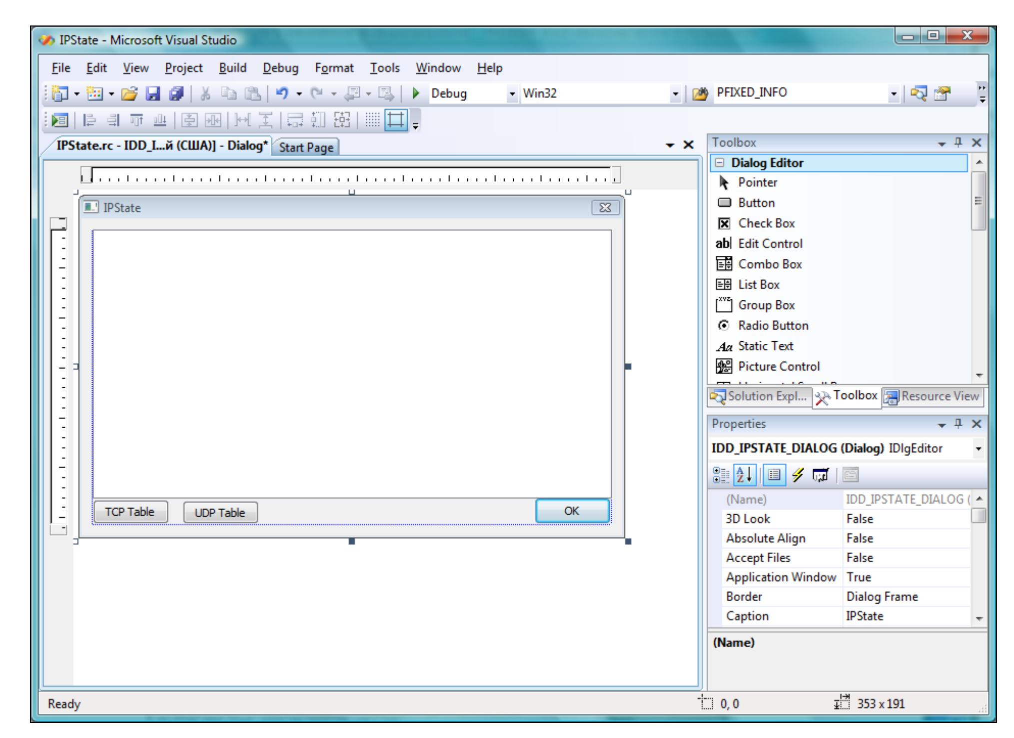


Рис. 6.3. Окно будущей программы IPState

По кнопке **TCP Table** должен выполняться код из листинга 6.3.

Листинг 6.3. Получение информации о TCP-портах

```
void CIPStateDlg::OnBnClickedButton1()
{
    DWORD dwStatus = NO_ERROR;
    PMIB_TCPTABLE pTcpTable = NULL;
    DWORD dwActualSize = 0;

    dwStatus = GetTcpTable(pTcpTable, &dwActualSize, TRUE);

    pTcpTable = (PMIB_TCPTABLE) malloc(dwActualSize);
    assert(pTcpTable);

    dwStatus = GetTcpTable(pTcpTable, &dwActualSize, TRUE);
    if (dwStatus != NO_ERROR)
```

```
{
    AfxMessageBox("Couldn't get tcp connection table.");
    free(pTcpTable);
    return;
}

CString strState;
struct in_addr inadLocal, inadRemote;
DWORD dwRemotePort = 0;
char szLocalIp[1000];
char szRemIp[1000];

if (pTcpTable != NULL)
{
    lList.AddString("=================================");
    lList.AddString("TCP table:");
    for (int i = 0; i < pTcpTable->dwNumEntries; i++)
    {
        dwRemotePort = 0;
        switch (pTcpTable->table[i].dwState)
        {
        case MIB_TCP_STATE_LISTEN:
            strState="Listen";
            dwRemotePort = pTcpTable->table[i].dwRemotePort;
            break;
        case MIB_TCP_STATE_CLOSED:
            strState="Closed"; break;
        case MIB_TCP_STATE_TIME_WAIT:
            strState="Time wait"; break;
        case MIB_TCP_STATE_LAST_ACK:
            strState="Last ACK"; break;
        case MIB_TCP_STATE_CLOSING:
            strState="Closing"; break;
        case MIB_TCP_STATE_CLOSE_WAIT:
            strState="Close Wait"; break;
        case MIB_TCP_STATE_FIN_WAIT1:
            strState="FIN wait"; break;
        case MIB_TCP_STATE_ESTAB:
            strState="EStab"; break;
        case MIB_TCP_STATE_SYN_RCVD:
            strState="SYN Received"; break;
        case MIB_TCP_STATE_SYN_SENT:
            strState="SYN Sent"; break;
```

```
            case MIB_TCP_STATE_DELETE_TCB:
                strState="Delete"; break;
            }
            inadLocal.s_addr = pTcpTable->table[i].dwLocalAddr;
            inadRemote.s_addr = pTcpTable->table[i].dwRemoteAddr;
            strcpy(szLocalIp, inet_ntoa(inadLocal));
            strcpy(szRemIp, inet_ntoa(inadRemote));

            char prtStr[1000];
            sprintf(prtStr, "Loc Addr %1s; Loc Port %1u; Rem Addr %1s;
                              Rem Port %1u; State %s;",
            szLocalIp,  ntohs((unsigned short)(0x0000FFFF & pTcpTable-
                               >table[i].dwLocalPort)),
            szRemIp, ntohs((unsigned short)(0x0000FFFF & dwRemotePort)),
                strState);
            lList.AddString(prtStr);
        }
    }
    free(pTcpTable);
}
```

У функции `GetTcpTable` три параметра:

- структура типа `PMIB_TCPTABLE`;
- размер структуры, указанной в качестве первого параметра;
- признак сортировки (если `TRUE`, то таблица будет отсортирована по номеру порта, иначе данные будут представлены в перемешанном виде).

Если в качестве первых двух параметров указать нулевое значение, то на выходе во втором параметре будет получен необходимый размер для хранения структур `PMIB_TCPTABLE`. Этот прием мы уже использовали в *главе 5* при работе с параметрами сети.

Память определенного размера выделяется функцией `malloc`, не обязательно в глобальной области.

Повторный вызов функции `GetTcpTable` позволяет через первый параметр (переменная `pTcpTable` типа `PMIB_TCPTABLE`) получить данные о состоянии всех TCP-портов. Их количество находится в параметре `dwNumEntries` структуры `pTcpTable`. Информацию об определенном порте можно узнать из параметра `table[i]`, где `i` — номер порта. Этот параметр тоже является структурой, и в нем нас интересуют следующие элементы:

- `dwState` — состояние порта, различные значения (`MIB_TCP_STATE_LISTEN`, `MIB_TCP_STATE_CLOSED` и т. д.). Список всех значений можно найти в коде

программы или в справочной системе. Назначение констант просто определить, достаточно только внимательно посмотреть на код из листинга 6.3;

- `dwLocalPort` — локальный порт;
- `dwRemotePort` — удаленный порт;
- `dwLocalAddr` — локальный адрес;
- `dwRemoteAddr` — удаленный адрес.

В примере запускается бесконечный цикл, который перебирает все записи из параметра `table`, и информация добавляется в список `List Box`. Результат выполнения в моей системе вы можете увидеть на рис. 6.4.

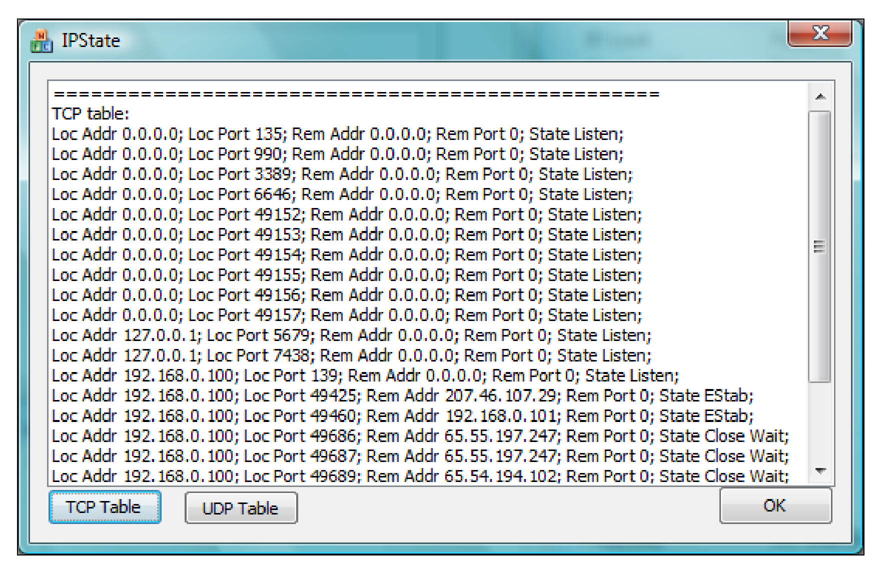


Рис. 6.4. Результат работы программы IPState

Для правильной компиляции программы в начале модуля надо подключить три заголовочных файла:

```
#include <iphlpapi.h>
#include <assert.h>
#include <winsock2.h>
```

В свойствах проекта, в разделе **Linker/Input** в пункте **Additional Dependencies** нужно добавить две библиотеки `IPHlpApi.lib` и `ws2_32.lib` (рис. 6.5).

Для получения таблицы UDP-портов используется функция `GetUdpTable`. Она работает аналогично, но узнать можно только локальный адрес и локальный

порт, потому что протокол UDP не устанавливает соединения, и нет сведений об удаленном компьютере.

Давайте добавим в программу кнопку **UDP Table** (листинг 6.4).

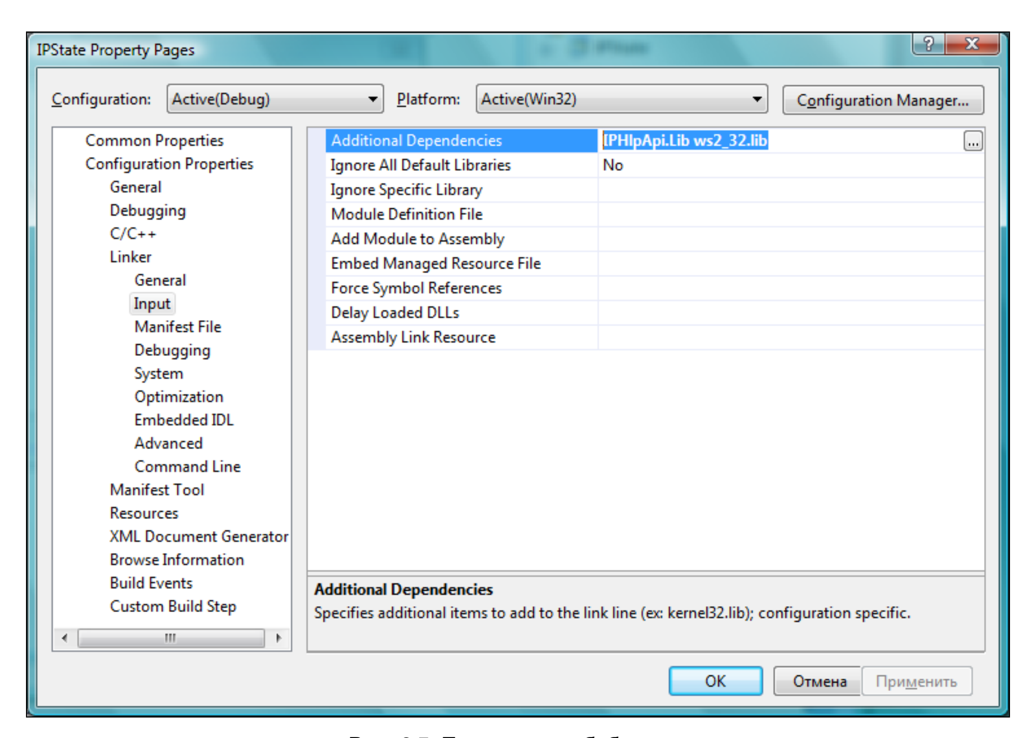


Рис. 6.5. Подключение библиотек

Листинг 6.4. Получение таблицы состояния UDP-портов

```
void CIPStateDlg::OnBnClickedButton2()
{
    DWORD dwStatus = NO_ERROR;
    PMIB_UDPTABLE pUdpTable = NULL;
    DWORD dwActualSize = 0;

    dwStatus = GetUdpTable(pUdpTable, &dwActualSize, TRUE);

    pUdpTable = (PMIB_UDPTABLE) malloc(dwActualSize);
    assert(pUdpTable);

    dwStatus = GetUdpTable(pUdpTable, &dwActualSize, TRUE);
```

```
        if (dwStatus != NO_ERROR)
    {
        AfxMessageBox("Couldn't get UDP connection table.");
        free(pUdpTable);
        return;
    }

    struct in_addr inadLocal;
    if (pUdpTable != NULL)
    {
        lList.AddString("=============");
        lList.AddString("UDP table:");
        for (int i = 0; i < pUdpTable->dwNumEntries; ++i)
        {
            inadLocal.s_addr = pUdpTable->table[i].dwLocalAddr;

            char prtStr[1000];
            sprintf(prtStr, "Loc Addr %1s; Loc Port %1u",
                inet_ntoa(inadLocal),
                ntohs((unsigned short)(0x0000FFFF & pUdpTable-
                         >table[i].dwLocalPort)));
            lList.AddString(prtStr);
        }
    }
    free(pUdpTable);
}
```

Код получения информации о UDP похож на тот, что использовался для протокола TCP, и вам не составит труда разобраться в происходящем.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter6\IPState.

6.4. DHCP-сервер

Если в вашей сети используется DHCP-сервер, то нельзя использовать удаление и добавление IP-адреса по образцу в *главе 5*. В этом случае адрес выдается и освобождается DHCP-сервером, а не вручную, иначе могут возникнуть проблемы и конфликты с другими компьютерами.

DHCP-адреса не удаляются из системы, а освобождаются. В этом случае сервер сможет отдать высвобожденный адрес другому компьютеру, если он же-

стко не привязан к определенному сетевому интерфейсу. Для освобождения используется функция `IpReleaseAddress`, которой надо передать нужный адаптер. Для получения адреса используется функция `IpRenewAddress`, которой также следует указать адаптер, нуждающийся в новом адресе.

Рассмотрю использование функций на примере. Для этого создадим новое MFC-приложение. Главное окно вы можете увидеть на рис. 6.6. Для определения адаптера, нуждающегося в удалении, нужно знать его индекс. Для этого внизу окна расположен элемент управления `List Box`, в котором будет отображаться список установленных интерфейсов. Вывод списка адаптеров будет происходить по кнопке **List Adapters**. Код аналогичен коду из листинга 5.2, где также выводилась информация об установленных адаптерах.

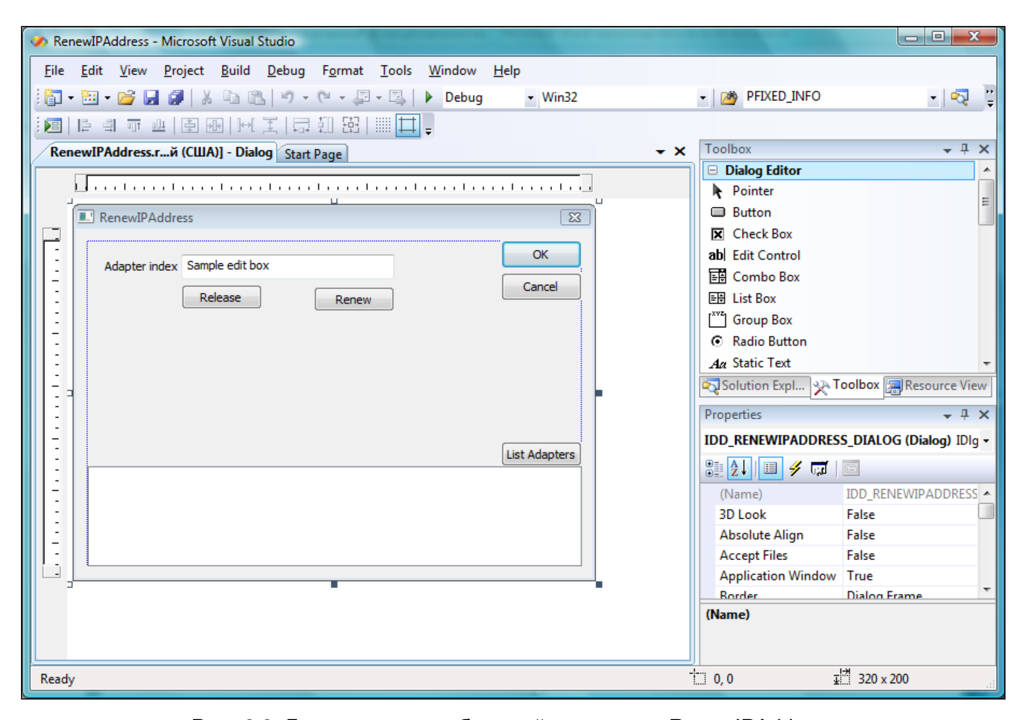


Рис. 6.6. Диалоговое окно будущей программы RenewIPAddress

По кнопке **Release** освобождается IP-адрес. Код, который должен выполняться, приведен в листинге 6.5.

Листинг 6.5. Освобождение IP-адреса

```
void CRenewIPAddressDlg::OnBnClickedButton2()
{
    char sAdaptIndex[20];
```

```
    int iIndex;
    sAdapterIndex.GetWindowText(sAdaptIndex, 20);
    iIndex = atoi(sAdaptIndex);

    DWORD InterfaceInfoSize = 0;
    PIP_INTERFACE_INFO pInterfaceInfo;

    if (GetInterfaceInfo(NULL, &InterfaceInfoSize) !=
           ERROR_INSUFFICIENT_BUFFER)
    {
        AfxMessageBox("Error sizing buffer");
        return;
    }

    if ((pInterfaceInfo = (PIP_INTERFACE_INFO) GlobalAlloc(GPTR,
           InterfaceInfoSize)) == NULL)
    {
        AfxMessageBox("Can't allocate memory");
        return;
    }

    if (GetInterfaceInfo(pInterfaceInfo, &InterfaceInfoSize) != 0)
    {
        AfxMessageBox("GetInterfaceInfo failed");
        return;
    }

    for (int i = 0; i < pInterfaceInfo->NumAdapters; i++)
        if (iIndex == pInterfaceInfo->Adapter[i].Index)
        {
            if (IpReleaseAddress(&pInterfaceInfo->Adapter[i]) != 0)
            {
                AfxMessageBox("IpReleaseAddress failed");
                return;
            }
            break;
        }
}
```

В поле ввода на главном окне пользователь указывает индекс адаптера. Теперь надо только пролистать все интерфейсы, и если какой-либо из них принадлежит адаптеру, то вызвать для него функцию `IpReleaseAddress`.

Для получения списка интерфейсов используется функция `GetInterfaceInfo`, которая работает так же, как и уже знакомая вам `GetAdaptersInfo`. При первом

вызове определяется необходимый размер памяти для хранения всей информации, после чего выделяется эта память, и функция вызывается снова.

Затем запускается цикл перебора всех полученных интерфейсов. Если интерфейс указанного адаптера найден, то освобождается адрес.

Получение адреса происходит подобным образом. В листинге 6.6 показан код кнопки **Renew**.

Листинг 6.6. Запрос нового IP-адреса

```
void CRenewIPAddressDlg::OnBnClickedButton3()
{
    char sAdaptIndex[20];
    int iIndex;
    sAdapterIndex.GetWindowText(sAdaptIndex, 20);
    iIndex = atoi(sAdaptIndex);

    DWORD InterfaceInfoSize = 0;
    PIP_INTERFACE_INFO pInterfaceInfo;

    if (GetInterfaceInfo(NULL, &InterfaceInfoSize) !=
              ERROR_INSUFFICIENT_BUFFER)
    {
        AfxMessageBox("Error sizing buffer");
        return;
    }

    if ((pInterfaceInfo = (PIP_INTERFACE_INFO) GlobalAlloc(GPTR,
            InterfaceInfoSize)) == NULL)
    {
        AfxMessageBox("Can't allocate memory");
        return;
    }

    if (GetInterfaceInfo(pInterfaceInfo, &InterfaceInfoSize) != 0)
    {
        AfxMessageBox("GetInterfaceInfo failed");
        return;
    }

    for (int i = 0; i < pInterfaceInfo->NumAdapters; i++)
        if (iIndex == pInterfaceInfo->Adapter[i].Index)
        {
```

```
            if (IpRenewAddress(&pInterfaceInfo->Adapter[i]) != 0)
            {
                AfxMessageBox("IpRenewAddress failed");
                return;
            }
            break;
        }
}
```

Код получения нового адреса аналогичен освобождению (см. листинг 6.5). Разница только в том, что в данном случае вызывается функция `IpRenewAddress`.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter6\RenewIPAddress.

6.5. Протокол ICMP

Я уже говорил о том, что протокол IP не обеспечивает надежность передачи данных, поэтому нельзя узнать о целостности принимаемых данных. Но с помощью протокола ICMP (Internet Control Message Protocol, интернет-протокол управляющих сообщений) можно узнать, достиг ли пакет адресата. Пакеты ICMP отправляются в тех случаях, когда адресат недоступен, буфер шлюза переполнен или недостаточен для отправки сообщения либо адресат требует передать данные по более короткому маршруту.

Протокол ICMP был разработан для того, чтобы информировать о возникающих проблемах во время приема/передачи и повысить надежность передачи информации по IP-протоколу, который изначально ненадежен. Но и на ICMP надеяться нельзя, потому что данные могут не дойти до адресата (заблудиться в сети), а вы не получите никаких сообщений. Именно поэтому используют протоколы более высокого уровня (например, TCP), имеющие свои методы обеспечения надежности.

Если вы хотите создать свой протокол на основе IP, то можете использовать сообщения ICMP для обеспечения определенной надежности, когда шлюз или компьютер не может обработать пакет. Но в случае обрыва, например, никаких сообщений не будет, потому что системы подтверждений в протоколе IP нет.

Из-за малой надежности протокола ICMP программисты его редко используют в тех целях, для которых он создавался (для контроля доставки данных). Но у него есть другое предназначение: если отправить компьютеру ICMP-пакет, и он дойдет до адресата, то тот должен ответить. Таким способом можно легко

проверить связь с удаленным компьютером. Именно таким образом реализованы программы Ping.

Для теста связи без использования ICMP нужно знать открытый порт на удаленном компьютере, чтобы попытаться соединиться с ним. Если связь прошла успешно, то компьютер доступен. Не зная порта, можно просканировать весь диапазон, но это займет слишком много времени. Протокол ICMP позволяет избежать этой процедуры.

В некоторых сетях на все машины ставят брандмауэры, которые запрещают протокол ICMP, и в этом случае администраторы открывают эхо-сервер (такой сервер получает данные и отвечает пакетом с этими же данными) и тестируют соединение через него. Такой способ хорош, но может возникнуть ситуация, когда связь есть, но эхо-сервер завис или его просто выключили. Обычно же ICMP-пакеты разрешены и работают нормально.

В теории все прекрасно, но на практике есть одна сложность: ICMP-протокол использует пакеты, отличные от TCP или UDP, поддержка которых есть в WinSock. Как же тогда отправить пакет с управляющим сообщением? Нужно самостоятельно формировать пакет необходимого формата. Есть еще один вариант — воспользоваться библиотекой icmp.dll, которая входит в состав Windows и упрощает работу с этим протоколом. Мы пойдем более сложным, но очень интересным путем.

В WinSock1 не было возможности доступа напрямую к данным пакета. Функция `select` в качестве второго параметра (тип спецификации) могла принимать только значения `SOCK_STREAM` (для TCP-протокола) или `SOCK_DGRAM` (для UDP-протокола), и я об этом говорил. В WinSock2 появилась поддержка RAW-сокетов низкоуровневого доступа к пакетам. Чтобы создать такой сокет (сырой), при вызове функции `socket` в качестве второго параметра нужно указать `SOCK_RAW`.

Рассмотрю, как программно реализована программа типа Ping. Это поможет вам понять, как работать с RAW-сокетами и как проверять связь с помощью ICMP-протокола. Создайте новое MFC-приложение. Главное диалоговое окно будущей программы можно увидеть на рис. 6.7.

В строке **Host** будет вводиться имя или IP-адрес компьютера, с которым надо проверить связь. По щелчку на кнопке **Ping** будут отправляться и приниматься ICMP-пакеты и выводиться результат в нижней части окна (листинг 6.7).

Листинг 6.7. Использование пакетов ICMP

```
void CPingerDlg::OnBnClickedButton1()
{
    SOCKET          rawSocket;
    LPHOSTENT lpHost;
```

```
struct    sockaddr_in sDest;
struct    sockaddr_in sSrc;
DWORD         dwElapsed;
int       iRet;
CString str;

WSADATA       wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    AfxMessageBox("Can't load WinSock");
    return;
}

// Creation socket (Создание сокета)
rawSocket = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
if (rawSocket == SOCKET_ERROR)
{
   AfxMessageBox("Socket error");
   return;
}

// Lookup host (Поиск хоста)
char strHost[255];
edHost.GetWindowText(strHost, 255);
lpHost = gethostbyname(strHost);
if (lpHost == NULL)
{
    AfxMessageBox("Host not found");
    return;
}

// Socket's address (Адрес сокета)
sDest.sin_addr.s_addr = *((u_long FAR *) (lpHost->h_addr));
sDest.sin_family = AF_INET;
sDest.sin_port = 0;

str.Format("Pinging %s [%s]",
                  strHost, inet_ntoa(sDest.sin_addr));

lMessages.AddString(str);

// Send ICMP echo request (Посылка эхо-запроса ICMP)
static ECHOREQUEST echoReq;
```

```
    echoReq.icmpHdr.Type = ICMP_ECHOREQ;
    echoReq.icmpHdr.Code = 0;
    echoReq.icmpHdr.ID   = 0;
    echoReq.icmpHdr.Seq  = 0;
    echoReq.dwTime = GetTickCount();
    FillMemory(echoReq.cData, 64, 80);
    echoReq.icmpHdr.Checksum = CheckSum((u_short *)&echoReq,
                 sizeof(ECHOREQUEST));

    // Send the echo request  (Отправка эхо-запроса)
    sendto(rawSocket, (LPSTR)&echoReq, sizeof(ECHOREQUEST),
                  0, (LPSOCKADDR)&sDest, sizeof(SOCKADDR_IN));

    struct timeval tOut;
    fd_set readfds;
    readfds.fd_count = 1;
    readfds.fd_array[0] = rawSocket;
    tOut.tv_sec = 1;
    tOut.tv_usec = 0;

    iRet=select(1, &readfds, NULL, NULL, & tOut);

    if (!iRet)
    {
        lMessages.AddString("Request Timed Out");
    }
    else
    {
        // Receive reply (Получение ответа)
        ECHOREPLY echoReply;
        int nRet;
        int nAddrLen = sizeof(struct sockaddr_in);

        iRet = recvfrom(rawSocket, (LPSTR)&echoReply,
                 sizeof(ECHOREPLY), 0, (LPSOCKADDR)&sSrc, &nAddrLen);

        if (iRet == SOCKET_ERROR)
               AfxMessageBox("Recvfrom Error");

        // Calculate time (Расчет времени)
        dwElapsed = GetTickCount() — echoReply.echoRequest.dwTime;
        str.Format("Reply from: %s: bytes=%d time=%ldms TTL=%d",
                       inet_ntoa(sSrc.sin_addr), 64, dwElapsed,
                       echoReply.ipHdr.TTL);
```

```
        lMessages.AddString(str);
    }

    iRet = closesocket(rawSocket);
    if (iRet == SOCKET_ERROR)
        AfxMessageBox("Closesocket error");

    WSACleanup();
}
```

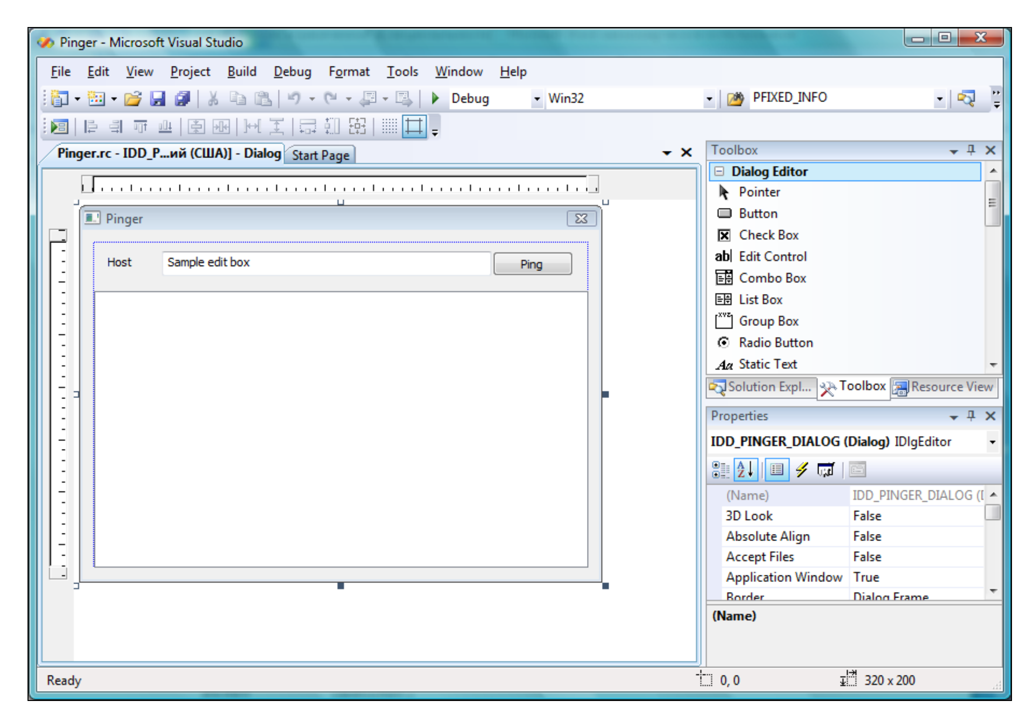


Рис. 6.7. Диалоговое окно будущей программы Pinger

Прежде чем работать с сетью, необходимо загрузить библиотеку WinSock с помощью функции `WSAStartup`. Работа с RAW-сокетами не исключение, но загружать надо библиотеку WinSocket2, потому что в первой версии нет необходимых возможностей.

После этого можно создавать сокет с помощью функции `socket` со следующими параметрами:

- семейство протокола — `AF_INET` (как всегда);
- спецификация — `SOCK_RAW` для использования RAW-сокетов;
- протокол — `IPPROTO_ICMP`.

Для отправки пакета с данными компьютеру необходимо знать его адрес. Если пользователь ввел символьное имя, то надо определить IP-адрес по имени с помощью функции `gethostbyname`.

После этого, как и в случае с другими протоколами, заполняется структура типа `sockaddr_in`, содержащая адрес компьютера, с которым нужно соединяться. ICMP-запрос не будет использовать портов, поэтому параметр `Port` установлен в `0`.

Затем заполняется структура типа `ECHOREQUEST`. Эта структура является пакетом, который будет отправляться в сеть. Если при использовании протоколов TCP или UDP необходимо только указать данные, которые подлежат отправке, то в случае с ICMP нужно формировать полный пакет, который будет отправлен через IP-протокол. Структура `ECHOREQUEST` имеет вид пакета и выглядит следующим образом:

```
typedef struct tagECHOREQUEST
{
       ICMPHDR icmpHdr;
       DWORD   dwTime;
       char    cData[64];
}ECHOREQUEST, *PECHOREQUEST;
```

Параметр `icmpHdr` — это заголовок пакета, который необходимо самостоятельно заполнить, а параметр `cData` содержит отправляемые данные. В нашем случае будут отправляться пакеты по 64 байта, поэтому объявлен массив из 64 символов. В программе весь массив заполняется символом с кодом 80 с помощью функции `FillChar`. Для программы Ping не имеет значения, какие отправлять данные, потому что главное — проверить возможность связи с удаленным компьютером.

Параметр `dwTime` — это время, которое можно использовать на свое усмотрение. По нему чаще всего определяют время прохождения пакета.

Заголовок формируется в зависимости от принимаемого или отправляемого сообщения. Так как я для примера выбрал программу типа Ping, то буду рассматривать необходимые для нее данные. Более подробное описание протокола ICMP вы можете найти в документе RFC792 по адресу **http://info.internet.isi.edu/in-notes/rfc/files/rfc792.txt**. Заголовок (параметр `icmpHdr`) — это структура следующего вида:

```
typedef struct tagICMPHDR
{
       u_char Type;
       u_char Code;
       u_short Checksum;
```

```
        u_short ID;
        u_short Seq;
        char Data;
}ICMPHDR, *PICMPHDR;
```

Рассмотрим эти параметры:

- `Type` — тип пакета. В нашем случае это `ICMP_ECHOREQ`, который означает эхо-запрос ответа (сервер должен вернуть те же данные, которые принял). При ответе этот параметр должен быть нулевым;
- `Code` — не используется в эхо-запросах и должен равняться нулю;
- `Checksum` — контрольная сумма. RFC не накладывает жестких требований на алгоритм, и он может быть изменен. В данной программе я использовал упрощенный алгоритм, который вы можете увидеть в листинге 6.8;
- `ID` — идентификатор. Для эхо-запроса должен быть обнулен, но может содержать и другие значения;
- `Seq` — номер очереди, который должен быть обнулен, если код равен нулю.

Листинг 6.8. Функция подсчета контрольной суммы

```
u_short CheckSum(u_short *addr, int len)
{
        register int nleft = len;
        register u_short answer;
        register int sum = 0;

        while( nleft > 1 )  {
                sum += *addr++;
                nleft -= 1;
        }

        sum += (sum >> 16);
        answer = ~sum;
        return (answer);
}
```

После формирования пакета он отправляется с помощью функции `sendto`, потому что в качестве транспорта используется IP-протокол, который не поддерживает соединение как TCP, и по своей работе схож с UDP-протоколом.

Для ожидания ответа используется функция `select`, с помощью которой ждем в течение секунды возможности чтения с сокета. Если за это время ответ не

получен, считается, что удаленный компьютер недоступен. Иначе читается пакет данных. В принципе, ответ уже известен: связь между компьютерами есть, и читать пакет не обязательно, но я сделаю это, чтобы вы увидели весь цикл приема/передачи сообщений через RAW-сокеты. В реальном приложении чтение пакета необходимо, чтобы удостовериться в том, что получен пакет от того компьютера, которому отправлены данные (возможно, что совершенно другой компьютер посылал данные). Чтение пакета необходимо и в том случае, когда пингуется асинхронно сразу несколько компьютеров.

Для чтения пакета используется функция `recvfrom`, как и при работе с UDP-протоколом. Если при отправке посылается пакет данных в виде структуры `ECHOREQUEST`, то при чтении принимается пакет типа `ECHOREPLY`, который выглядит следующим образом:

```
typedef struct tagECHOREPLY
{
       IPHDR         ipHdr;
       ECHOREQUEST   echoRequest;
       char          cFiller[256];
}ECHOREPLY, *PECHOREPLY;
```

Первый параметр — это заголовок пришедшего пакета. Второй параметр — это заголовок пакета с данными, который был послан.

Заголовок принятого пакета отличается от отправленного и выглядит следующим образом:

```
typedef struct tagIPHDR
{
       u_char  VIHL;
       u_char  TOS;
       short   TotLen;
       short   ID;
       short   FlagOff;
       u_char  TTL;
       u_char  Protocol;
       u_short Checksum;
       struct  in_addr iaSrc;
       struct  in_addr iaDst;
}IPHDR, *PIPHDR;
```

Это не что иное, как заголовок протокола IP.

Все необходимые структуры должны быть описаны в заголовочном файле. Для приведенного примера я описал все в файле PingerDlg.h.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter6\Pinger.

6.6. Определение пути пакета

Как можно определить путь пакета, по которому он идет от нас до адресата? Каждый пакет имеет поле TTL (Time To Leave, время жизни). Каждый маршрутизатор уменьшает значение поля на единицу, и когда оно становится равным нулю, пакет считается заблудившимся, и маршрутизатор возвращает ICMP-сообщение об ошибке. Использование этого поля еще упрощает проблему.

Надо направить пакет со временем жизни, равным 1. Первый же маршрутизатор уменьшит значение на 1 и увидит 0. Это заставит его вернуть ICMP-сообщение об ошибке, по которому можно узнать первый узел, через который проходит пакет. Затем отсылается пакет со временем жизни, равным 2, и определяется второй маршрутизатор (первый пропустит пакет, а второй вернет ICMP-сообщение). Таким образом можно действовать, пока не достигнем адресата.

Стало быть, есть возможность узнать не только маршрут, но и время отклика каждого маршрутизатора, что позволяет определить слабое звено на пути следования пакета. Связь с каждым из устройств на пути пакета может изменяться в зависимости от нагрузки, поэтому желательно сделать несколько попыток соединения, чтобы определить среднее время отклика.

Конечно же, первый пакет может пойти одним маршрутом, а второй — другим, но чаще всего все пакеты следуют по одному и тому же маршруту.

Сейчас я покажу простейший пример определения пути следования маршрута. Но только ICMP-пакеты я буду посылать не через RAW-сокеты, а через библиотеку icmp.dll. В этой библиотеке есть все необходимые функции для создания сокета и отправки пакета. Нужно только указать адрес, содержимое пакета и его параметры, а все остальное сделают за вас. Таким образом, вы научитесь пользоваться библиотекой icmp.dll и сможете выяснить путь прохождения пакета.

Создайте новое MFC-приложение TraceRote. В главном окне вам понадобится одна строка ввода для указания адреса компьютера, связь с которым необходимо проверить, один компонент типа `List Box` для отображения информации и кнопка (например, **Trace**), по которой будет пинговаться удаленный компьютер (листинг 6.9).

Листинг 6.9. Определение пути следования пакета

```
void CTraceRouteDlg::OnBnClickedButton1()
{
       WSADATA wsa;
       if (WSAStartup(MAKEWORD(1, 1), &wsa) != 0)
       {
              AfxMessageBox("Can't load a correct version of WinSock");
              return;
       }

       hIcmp = LoadLibrary("ICMP.DLL");
       if (hIcmp == NULL)
       {
              AfxMessageBox("Can't load ICMP DLL");
              return;
       }

       pIcmpCreateFile = (lpIcmpCreateFile)
                GetProcAddress(hIcmp,"IcmpCreateFile");
       pIcmpSendEcho = (lpIcmpSendEcho)
                GetProcAddress(hIcmp,"IcmpSendEcho" );
       pIcmpCloseHandle = (lpIcmpCloseHandle)
                GetProcAddress(hIcmp,"IcmpCloseHandle");

       in_addr  Address;
       if (pIcmpCreateFile == NULL)
       {
              AfxMessageBox("ICMP library error");
              return;
       }

       char chHostName[255];
       edHostName.GetWindowText(chHostName, 255);
       LPHOSTENT hp = gethostbyname(chHostName);
       if (hp== NULL)
       {
           AfxMessageBox("Host not found");
           return;
       }
       unsigned long       addr;
       memcpy(&addr, hp->h_addr, hp->h_length);
```

```
BOOL bReachedHost = FALSE;
for (UCHAR i=1; i<=50 && !bReachedHost; i++)
{
    Address.S_un.S_addr = 0;

    int iPacketSize=32;
    int iRTT;

    HANDLE hIP = pIcmpCreateFile();
    if (hIP == INVALID_HANDLE_VALUE)
    {
        AfxMessageBox("Could not get a valid ICMP handle");
        return;
    }

    unsigned char* pBuf = new unsigned char[iPacketSize];
    FillMemory(pBuf, iPacketSize, 80);

    int iReplySize = sizeof(ICMP_ECHO_REPLY) + iPacketSize;
    unsigned char* pReplyBuf = new unsigned char[iReplySize];
    ICMP_ECHO_REPLY* pEchoReply = (ICMP_ECHO_REPLY*) pReplyBuf;

     IP_OPTION_INFORMATION ipOptionInfo;
     ZeroMemory(&ipOptionInfo, sizeof(IP_OPTION_INFORMATION));
     ipOptionInfo.Ttl = i;

     DWORD nRecvPackets = pIcmpSendEcho(hIP, addr, pBuf,
              iPacketSize, &ipOptionInfo, pReplyBuf,
              iReplySize, 30000);

     if (nRecvPackets != 1)
     {
         AfxMessageBox("Can't ping host");
         return;
     }
     Address.S_un.S_addr = pEchoReply->Address;
     iRTT = pEchoReply->RoundTripTime;

     pIcmpCloseHandle(hIP);

     delete [] pReplyBuf;
     delete [] pBuf;

     char lpszText[255];
```

```
        hostent* phostent = NULL;
        phostent = gethostbyaddr((char *)&Address.S_un.S_addr, 4, PF_INET);

        if (phostent)
            sprintf(lpszText, "%d: %d ms [%s] (%d.%d.%d.%d)",
                  i, iRTT,
                  phostent->h_name, Address.S_un.S_un_b.s_b1,
                                    Address.S_un.S_un_b.s_b2,
                                    Address.S_un.S_un_b.s_b3,
                                    Address.S_un.S_un_b.s_b4);
        else
            sprintf(lpszText, "%d — %d ms (%d.%d.%d.%d)",
                  i, iRTT,
                  Address.S_un.S_un_b.s_b1,
                  Address.S_un.S_un_b.s_b2,
                  Address.S_un.S_un_b.s_b3,
                  Address.S_un.S_un_b.s_b4);

        lbMessages.AddString(lpszText);

        if (addr == Address.S_un.S_addr)
            bReachedHost = TRUE;
    }

    if (hIcmp)
    {
        FreeLibrary(hIcmp);
        hIcmp = NULL;
    }

    WSACleanup();
}
```

Несмотря на дополнительную библиотека icmp.dll, библиотеку WinSock надо загрузить в любом случае. К тому же будет использоваться функция `gethostbyname` для определения IP-адреса, если пользователь укажет символьное имя компьютера. В данном случае будет достаточно первой версии библиотеки, без RAW-сокетов. Таким образом, программа сможет работать и в Windows 98 (без WinSock 2.0).

После этого нужно загрузить динамическую библиотеку icmp.dll с помощью функции `LoadLibrary`. Она находится в папке windows/system (или windows/system32), поэтому не надо указывать полный путь. Программа без проблем найдет и загрузит библиотеку.

В библиотеке нас будут интересовать следующие процедуры:

- `IcmpCreateFile` — инициализация;
- `IcmpSendEcho` — отправка эхо-пакета;
- `IcmpCloseHandle` — закрытие ICMP.

Прежде чем посылать пакет, следует его проинициализировать с помощью функции `IcmpCreateFile`. По окончании работы с ICMP нужно вызвать функцию `IcmpCloseHandle`, чтобы закрыть его.

Теперь в заранее подготовленные переменные запоминаются адреса необходимых процедур из библиотеки:

```
pIcmpCreateFile=(lpIcmpCreateFile)GetProcAddress(hIcmp,"IcmpCreateFile");
pIcmpSendEcho = (lpIcmpSendEcho)GetProcAddress(hIcmp,"IcmpSendEcho");
pIcmpCloseHandle=(lpIcmpCloseHandle)GetProcAddress(hIcmp,"IcmpCloseHandle");
```

Если писать программу по всем правилам, то необходимо было бы проверить полученные адреса на равенство нулю. Если хотя бы один адрес функции нулевой, то она не найдена, и дальнейшее ее использование невозможно. Чаще всего такое бывает из-за неправильного написания имени функции. Но может случиться, когда программа загрузит другую библиотеку с таким же именем, в которой вообще нет таких функций. Чтобы этого не произошло, переменная `pIcmpCreateFile` (она должна содержать адрес функции `IcmpCreateFile`) проверяется на равенство нулю. Если да, то загрузилась ошибочная библиотека, и об этом выводится соответствующее сообщение. Остальные переменные не проверяются (в надежде на правильное написание).

Следующим этапом определяется адрес компьютера, путь к которому надо найти, и переводится в IP-адрес. Если в этот момент произошла ошибка, то адрес указан неверно, и дальнейшее выполнение кода невозможно.

Вот теперь можно переходить к пингованию удаленного компьютера. Так как может возникнуть необходимость послать несколько пакетов с разным временем жизни, запускается цикл от 1 до 50. Использование в данном случае цикла `while`, который выполнялся бы, пока пинг не дойдет до нужного компьютера, не рекомендуется, т. к. появляется вероятность возникновения бесконечного цикла.

Внутри цикла инициализируется ICMP-пакет с помощью функции `IcmpCreateFile`. Результатом будет указатель на созданный объект, который понадобится при посылке эхо-пакета, поэтому он сохраняется в переменной `hIP` типа `HANDLE`:

```
HANDLE hIP = pIcmpCreateFile();
if (hIP == INVALID_HANDLE_VALUE)
```

```
{
      AfxMessageBox("Could not get a valid ICMP handle");
      return;
}
```

Если результат равен `INVALID_HANDLE_VALUE`, то во время инициализации произошла ошибка, и дальнейшее выполнение невозможно.

После этого выделяется буфер для данных, который, как и в случае с пингом, заполняется символом с кодом 80. Далее создается пакет типа `ICMP_ECHO_REPLY`, в котором возвращается информация, полученная от маршрутизатора или компьютера. Нужно также создать пакет типа `IP_OPTION_INFORMATION`, в котором указывается время жизни пакета (параметр `Ttl`).

Когда все подготовлено, можно отправлять ICMP-пакет с помощью функции `IcmpSendEcho`, у которой 8 параметров:

- указатель ICMP (получен во время инициализации);
- адрес компьютера;
- буфер с данными;
- размер пакета (с учетом объема посылаемых данных);
- IP-пакет с указанием времени жизни (на первом шаге он будет равен единице, потом двум и т. д.);
- буфер для хранения структуры типа `ICMP_ECHO_REPLY`, в которую будет записан результирующий пакет;
- размер буфера;
- время ожидания ответа.

В качестве результата функция возвращает количество принятых пакетов. В нашем случае он один. Если возвращаемое значение равно нулю, то маршрутизатор или компьютер не ответили ICMP-пакетом, и невозможно выяснить его параметры.

Время ответа можно получить из параметра `RoundTripTime` структуры `ICMP_ECHO_REPLY`, а адрес сетевого устройства, ответившего на запрос, — из параметра `Address`.

После работы не забывайте закрывать указатель на созданный ICMP с помощью `IcmpCloseHandle`.

Теперь можно выводить полученную информацию. Для удобства восприятия в программе реализован перевод IP-адреса в символьное имя с помощью функции `gethostbyaddr`.

У этой функции три параметра:

- IP-адрес компьютера, символьное имя которого надо определить;
- длина адреса;
- семейство протокола. От этого зависит формат предоставляемого адреса.

Далее проверяется: если поступил ответ от искомого компьютера, то цикл прерывается, иначе нужно увеличить на единицу время жизни пакета и повторить посылку ICMP-пакета:

```
if (addr == Address.S_un.S_addr)
       bReachedHost = TRUE;
```

По окончании работы нужно выгрузить из памяти библиотеку icmp.dll и освободить библиотеку WinSock:

```
if (hIcmp)
{
       FreeLibrary(hIcmp);
       hIcmp = NULL;
}
WSACleanup();
```

Запустите программу TraceRoute. На рис. 6.8 показано окно с результатами ее работы.

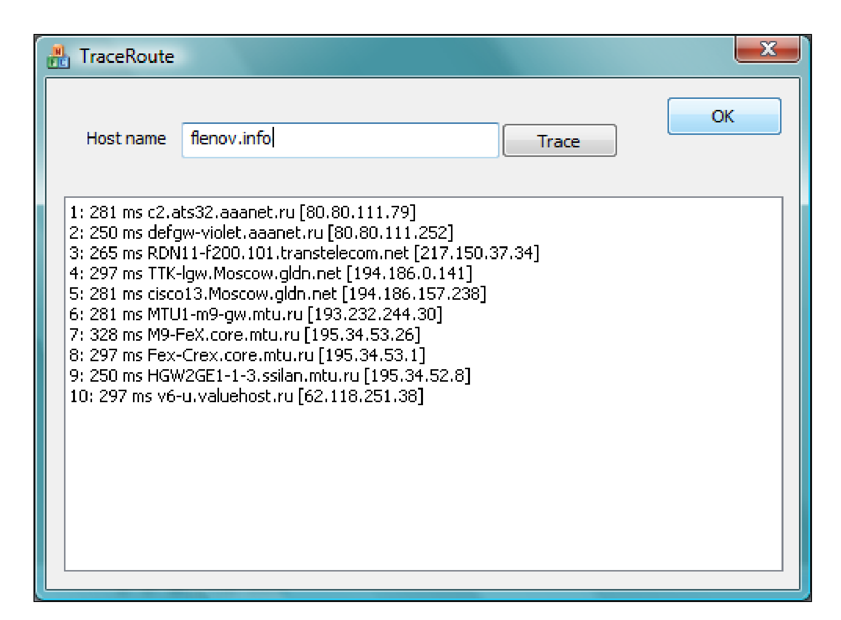


Рис. 6.8. Окно с результатом работы программы TraceRoute

Я постарался сделать пример простым, а логику прямолинейной, чтобы вам легче было разобраться. При этом если в процессе работы происходит ошибка, то на выходе из программы библиотеки icmp и winsock остаются загруженными. Самый простой способ избавиться от этого недостатка — производить загрузку библиотек при старте, а выгрузку — при выходе из программы, и если библиотеки не загружены, то можно отключать кнопки отправки пакета, чтобы пользователь не смог ими воспользоваться.

В Интернете можно найти заголовочные файлы для библиотеки icmp.dll, которые могут еще больше упростить этот пример. Но я не стал их использовать, чтобы ничего не ускользнуло от вашего внимания.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter6\TraceRoute.

6.7. ARP-протокол

Уже говорили о том, что перед обращением к компьютеру по локальной сети необходимо узнать его MAC-адрес. Для этого существует ARP-протокол, который по IP-адресу ищет MAC-адрес. Происходит это автоматически и незаметно для рядового пользователя, но иногда появляется возможность ручного управления таблицей ARP. В ОС Windows для этих целей есть утилита с одноименным названием `arp`, но она консольная, и работать с ней не очень удобно. Сейчас я покажу простую графическую утилиту, на примере которой и объясню функции работы с данным протоколом.

Создайте новое MFC-приложение на основе диалога. Внешний вид главного диалогового окна представлен на рис. 6.9. Здесь используются две строки ввода для указания IP- и MAC-адреса. По нажатию кнопки **Add** будет добавляться новая ARP-запись, в которой по указанному IP-адресу будет определяться MAC-адрес. По нажатию кнопки **Update** в списке `List Box` будет отображаться таблица ARP. Кнопка **Delete** будет использоваться для удаления записи из таблицы.

В обработчик события кнопки **Update** нужно написать код из листинга 6.10.

Листинг 6.10. Отображение таблицы ARP

```
void CARPApplicationDlg::OnBnClickedButton1()
{
        DWORD dwStatus;
        PMIB_IPNETTABLE pIpArpTab=NULL;
```

```
DWORD dwActualSize = 0;
GetIpNetTable(pIpArpTab, &dwActualSize, true);

pIpArpTab = (PMIB_IPNETTABLE) malloc(dwActualSize);
if (GetIpNetTable(pIpArpTab, &dwActualSize, true) != NO_ERROR)
{
       if (pIpArpTab)
              free (pIpArpTab);
       return;
}

DWORD i, dwCurrIndex;
char sPhysAddr[256], sType[256], sAddr[256];
PMIB_IPADDRTABLE pIpAddrTable = NULL;
char Str[255];

dwActualSize = 0;
GetIpAddrTable(pIpAddrTable, &dwActualSize, true);
pIpAddrTable = (PMIB_IPADDRTABLE) malloc(dwActualSize);
GetIpAddrTable(pIpAddrTable, &dwActualSize, true);

dwCurrIndex = -100;

for (i = 0; i < pIpArpTab->dwNumEntries; ++i)
{
       if (pIpArpTab->table[i].dwIndex != dwCurrIndex)
       {
              dwCurrIndex = pIpArpTab->table[i].dwIndex;


              struct in_addr in_ad;
              sAddr[0] = '\n';
              for (int i = 0; i < pIpAddrTable->dwNumEntries; i++)
              {
                     if (dwCurrIndex!=pIpAddrTable-
                         >table[i].dwIndex)
                            continue;

                     in_ad.s_addr = pIpAddrTable->table[i].dwAddr;
                     strcpy(sAddr, inet_ntoa(in_ad));
              }

              sprintf(Str,"Interface: %s on Interface 0x%X",
                      sAddr, dwCurrIndex);
```

```
                lbMessages.AddString(Str);
                lbMessages.AddString(" Internet Address | Physical
                                     Address | Type");
        }

            AddrToStr(pIpArpTab->table[i].bPhysAddr, pIpArpTab-
                     >table[i].dwPhysAddrLen, sPhysAddr);

        switch (pIpArpTab->table[i].dwType)
        {
        case 1:
            strcpy(sType,"Other");
            break;
        case 2:
            strcpy(sType,"Invalidated");
            break;
        case 3:
            strcpy(sType,"Dynamic");
            break;
        case 4:
            strcpy(sType,"Static");
            break;
        default:
            strcpy(sType,"");
        }

            struct in_addr in_ad;
            in_ad.s_addr = pIpArpTab->table[i].dwAddr;
            sprintf(Str, " %-16s | %-17s | %-11s", inet_ntoa(in_ad),
                          sPhysAddr, sType);
            lbMessages.AddString(Str);
        }

        free(pIpArpTab);
}
```

Посмотрите внимательно на код. Обратите внимание, что в данном случае не загружается библиотека WinSock. К проекту подключается заголовочный файл winsock.h, потому что используются типы данных, которые объявлены в нем. Но обращение к ним происходит только при компиляции. В данном случае не применяются библиотечные функции, необходимые на этапе выполнения.

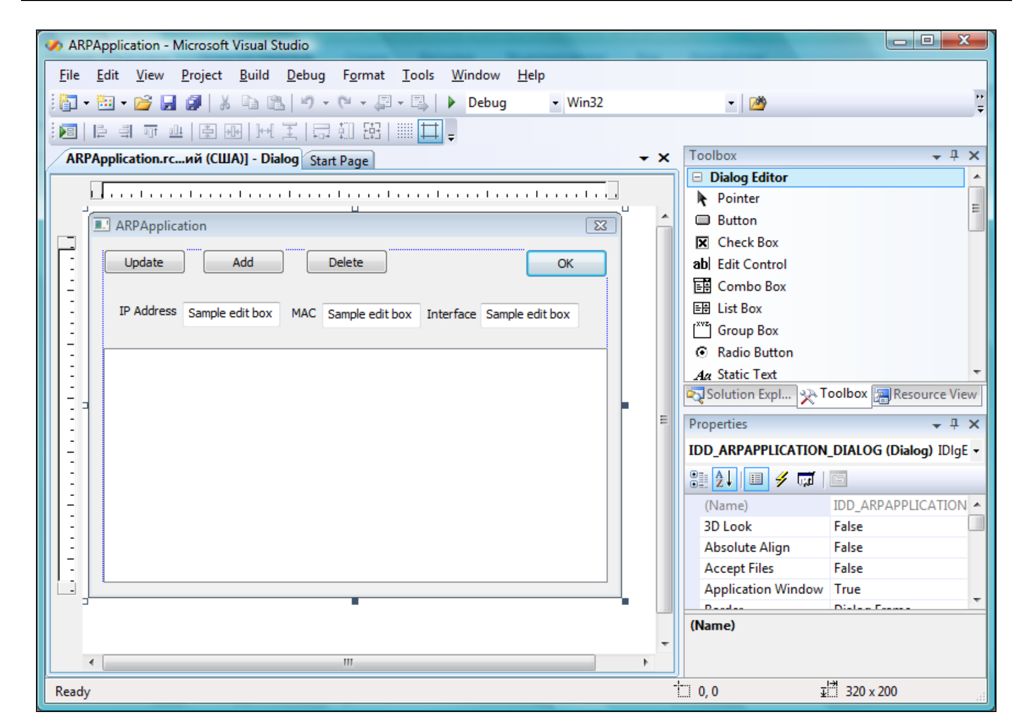


Рис. 6.9. Форма программы ARPApplication

Если вы добавите в программу код, который будет содержать хотя бы одну функцию из библиотеки WinSock (например, определение имени хоста по IP-адресу), то прежде чем ее использовать, необходимо будет загрузить библиотеку.

Первым делом из системы получена таблица соответствия IP-адресов их физическим MAC-адресам. Это и есть не что иное, как ARP-таблица. Функция для получения этой таблицы имеет следующий вид:

```
DWORD GetIpNetTable(
  PMIB_IPNETTABLE pIpNetTable,
  PULONG pdwSize,
  BOOL bOrder
);
```

ARP-таблица описывается следующими параметрами:

- указатель на структуру типа `MIB_IPNETTABLE`, в которой будет размещена таблица;
- размер структуры. Если он нулевой, то функция вернет объем памяти, требуемый для таблицы;
- сортировка (если `true`, то таблица будет упорядоченной).

Структура `MIB_IPNETTABLE`, которая задается в качестве первого параметра, имеет следующий вид:

```
typedef struct _MIB_IPNETTABLE {
       DWORD dwNumEntries;
       MIB_IPNETROW table[ANY_SIZE];
} MIB_IPNETTABLE, *PMIB_IPNETTABLE;
```

Первый параметр указывает на количество записей в таблице, а второй — это структура, содержащая данные таблицы:

```
typedef struct _MIB_IPNETROW {
       DWORD dwIndex;
       DWORD dwPhysAddrLen;
       BYTE bPhysAddr[MAXLEN_PHYSADDR];
       DWORD dwAddr;
       DWORD dwType;
} MIB_IPNETROW, *PMIB_IPNETROW;
```

Структура с данными таблицы описывается набором параметров:

- `dwIndex` — индекс адаптера;
- `dwPhysAddrLen` — длина физического адреса;
- `bPhysAddr` — физический адрес;
- `dwAddr` — IP-адрес;
- `dwType` — тип записи. В свою очередь может принимать такие значения:
  - `4` — статический (запись добавлена вручную с помощью функций, которые будут рассмотрены позже);
  - `3` — динамический (записи с адресами, которые получены автоматически с помощью протокола ARP. Они действительны в течение определенного времени, а потом автоматически уничтожаются);
  - `2` — неправильный (записи с ошибками);
  - `1` — другой.

В принципе, ARP-таблица уже создана. Если в компьютере установлена только одна сетевая карта, то этого достаточно, потому что все записи будут относиться к ней. Если установлены хотя бы две сетевые карты, то одна часть записей будет принадлежать первому интерфейсу, а другая — второму.

Чтобы картина была полной, необходимо показать, какие записи к какому интерфейсу относятся. Для этого есть индекс интерфейса в структуре `MIB_IPNETROW`, но он абсолютно ничего не скажет конечному пользователю. В сочетании с IP-адресом адаптера этот индекс станет более информативным.

А вот IP-адреса адаптера у нас пока нет. Чтобы его узнать, нужно получить таблицу соответствия IP-адресов адаптерам с помощью функции `GetIpAddrTable`. Функция похожа на `GetIpNetTable`:

```
DWORD GetIpAddrTable(
  PMIB_IPADDRTABLE pIpAddrTable,
  PULONG pdwSize,
  BOOL bOrder
);
```

Она имеет три параметра: указатель на структуру типа `MIB_IPADDRTABLE` (`pIpAddrTable`), размер структуры (`pdwSize`) и флаг сортировки (`bOrder`).

Первый параметр — это структура следующего вида:

```
typedef struct _MIB_IPADDRTABLE {
       DWORD dwNumEntries;
       MIB_IPADDRROW table[ANY_SIZE];
} MIB_IPADDRTABLE, *PMIB_IPADDRTABLE;
```

У этой структуры два параметра:

- `dwNumEntries` — количество структур, указанных во втором параметре;
- `table` — массив структур типа `MIB_IPADDRROW`.

Структура `MIB_IPADDRROW` описывается следующим образом:

```
typedef struct _MIB_IPADDRROW {
       DWORD dwAddr;
       DWORDIF_INDEX dwIndex;
       DWORD dwMask;
       DWORD dwBCastAddr;
       DWORD dwReasmSize;
       unsigned short unused1;
       unsigned short wType;
} MIB_IPADDRROW, *PMIB_IPADDRROW;
```

В этой структуре следующие параметры:

- `dwAddr` — IP-адрес;
- `dwIndex` — индекс адаптера, с которым связан IP-адрес;
- `dwMask` — маска для IP-адреса;
- `dwBCastAddr` — широковещательный адрес. Чаще всего это IP-адрес, в котором нулевое значение номера узла. Например, если у вас IP-адрес 192.168.4.7, то широковещательный адрес будет 192.168.4.0;
- `dwReasmSize` — максимальный размер получаемых пакетов;

- unused1 — зарезервировано;
- wType — тип адреса, может принимать следующие значения:
  - MIB_IPADDR_PRIMARY — основной IP-адрес;
  - MIB_IPADDR_DYNAMIC — динамический адрес;
  - MIB_IPADDR_DISCONNECTED — адрес в отключенном интерфейсе, например, если отсутствует сетевой кабель;
  - MIB_IPADDR_DELETED — адрес в процессе удаления;
  - MIB_IPADDR_TRANSIENT — временный адрес.

Когда получены необходимые данные, запускается цикл перебора всех строк в таблице ARP. Но прежде чем выводить на экран информацию о строке, надо проверить, к какому интерфейсу она относится.

При получении данных был выбран режим сортировки записей, поэтому можно надеяться, что вначале идут записи одного интерфейса, а потом другого. Поэтому перед циклом в переменную dwCurrIndex занесено значение –100. Интерфейса с таким номером точно не будет. На первом же шаге цикла будет видно, что запись из ARP-таблицы не относится к интерфейсу с номером –100, и необходимо вывести на экран IP-адрес сетевой карты, к которой относится эта запись. Для этого по параметру dwIndex ищется запись в таблице соответствия IP-адресов номерам интерфейса. Если запись найдена (а она должна быть найдена), то выводится заголовок таблицы, который будет выглядеть примерно так:

```
Interface: 192.168.1.100 on Interface 0x10000003
Internet Address | Physical Address | Type
```

Затем выводится информация из ARP-таблицы, пока не встретится запись, относящаяся к другому интерфейсу. Тогда снова выводится заголовок и т. д.

Запустите пример и посмотрите его в работе. Обратите внимание, что у вас может и не быть ARP-записей, потому что они существуют только при работе в локальной сети. При выходе в Интернет по модему протокол ARP не используется.

Если вы пока не знаете, как применить программу, то у меня уже были случаи, когда она оказалась незаменима. Допустим, что нужно узнать в локальной сети MAC-адрес компьютера, который находится от вас очень далеко. Можно пойти к этому компьютеру и посмотреть адрес с помощью утилиты ipconfig, а можно произвести следующие действия:

1. Выполнить программу Ping для проверки связи с удаленным компьютером. В этот момент отсылаются эхо-пакеты, которым также нужно знать MAC-адрес, и для этого задействуется ARP-протокол.

2. Запустить программу просмотра ARP-таблицы и там посмотреть MAC-адрес нужного компьютера.

Теперь посмотрите, как можно добавлять новые записи в таблицу ARP. Напоминаю, что все записи, добавленные программно, становятся статичными и не уничтожаются автоматически на протяжении всей работы ОС. По нажатию кнопки **Add** будет выполняться код из листинга 6.11.

Листинг 6.11. Добавление новой записи в таблицу ARP

```
void CARPApplicationDlg::OnBnClickedButton2()
{
       DWORD dwInetAddr = 0;
       char sPhysAddr[255];

       char sInetAddr[255], sMacAddr[255], sInterface[255];
       edIPAddress.GetWindowText(sInetAddr, 255);
       edMacAddress.GetWindowText(sMacAddr, 255);
       edInterface.GetWindowText(sInterface, 255);


       if (sInetAddr == NULL || sMacAddr == NULL || sInterface == NULL)
       {
              AfxMessageBox("Fill IP address, MAC address and
                             Interface");
              return;
       }

       dwInetAddr = inet_addr(sInetAddr);
       if (dwInetAddr == INADDR_NONE)
       {
              AfxMessageBox("Bad IP Address");
              return;
       }

       StrToMACAddr(sMacAddr, sPhysAddr);

       MIB_IPNETROW arpRow;
       sscanf(sInterface, "%X",&(arpRow.dwIndex));

       arpRow.dwPhysAddrLen = 6;
       memcpy(arpRow.bPhysAddr, sPhysAddr, 6);
       arpRow.dwAddr = dwInetAddr;
       arpRow.dwType = MIB_IPNET_TYPE_STATIC;
```

```
	if (SetIpNetEntry(&arpRow) != NO_ERROR)
		AfxMessageBox("Couldn't add ARP record");
}
```

Самое главное здесь — это функция `SetIpNetEntry`, которая добавляет новую ARP-запись и выглядит следующим образом:

```
DWORD SetIpNetEntry(
  PMIB_IPNETROW pArpEntry
);
```

В качестве единственного параметра функции указывается структура типа `MIB_IPNETROW`, которую мы уже использовали при получении данных ARP-таблицы. В этой структуре необходимо указать четыре параметра: интерфейс (`dwIndex`), MAC-адрес (`bPhysAddr`) и IP-адрес (`dwInetAddr`), запись которого надо добавить, и тип записи (в поле `dwType` значение `MIB_IPNET_TYPE_STATIC`). Остальные поля в этой функции не используются, и их заполнять не надо.

Теперь посмотрите на функцию удаления. По нажатию кнопки **Delete** выполняется код из листинга 6.12.

Листинг 6.12. Удаление записи из ARP-таблицы

```
void CARPApplicationDlg::OnBnClickedButton3()
{
	char sInetAddr[255],
		sInterface[255];

	edIPAddress.GetWindowText(sInetAddr, 255);
	edInterface.GetWindowText(sInterface, 255);

	if (sInetAddr == NULL || sInterface == NULL)
	{
		AfxMessageBox("Fill IP address and Interface");
		return;
	}

	DWORD dwInetAddr;
	dwInetAddr = inet_addr(sInetAddr);
	if (dwInetAddr == INADDR_NONE)
	{
		printf("IpArp: Bad Argument %s\n", sInetAddr);
		return;
	}
```

```
        MIB_IPNETROW arpEntry;

        sscanf(sInterface, "%X",&(arpEntry.dwIndex));
        arpEntry.dwAddr = dwInetAddr;

        if (DeleteIpNetEntry(&arpEntry) != NO_ERROR)
               AfxMessageBox("Couldn't delete ARP record");
}
```

Для удаления записи используется функция `DeleteIpNetEntry`, которая выглядит следующим образом:

```
DWORD DeleteIpNetEntry(
  PMIB_IPNETROW pArpEntry
);
```

У нее один параметр в виде структуры `PMIB_IPNETROW`, в которой нужно указывать только интерфейс и IP-адрес, запись которого надо удалить.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter6\ARPApplication.

ГЛАВА 7



Система безопасности

Эту главу я решил посвятить системе безопасности популярной и любимой мною ОС Windows. Тут много интересных вопросов, на которые нам предстоит ответить.

Мы будем рассматривать такие интересные темы, как работа с пользователями и группами. Вы узнаете, что скрывает от вас Windows XP Home Edition и Windows Vista Home Premium. А эти две системы реально кое-что инетересное скрывают от нас, о чем не пишут в пресс-релизах.

7.1. Пользователи ОС Windows

Управление пользователями в ОС Windows — не очень сложная тема, но очень интересная. Все функции, которые мы будем рассматривать в этой главе, относятся к Network Management SDK. Именно к такому файлу справки следует обращаться за дополнительной информацией. Описания функций находятся в файлах lm.h и lmaccess.h, а для компиляции понадобится еще и подключение дополнительно библиотеки Netapi32.lib, без которой сборка проекта будет невозможна.

7.1.1. Получение списка пользователей/групп

Начнем рассмотрение функций Netapi с определения списка пользователей и групп в системе. Вот тут необходимо заметить, что если вы используете версию Professional или Server, то у вас в разделе **Администрирование** Панели управления будет оснастка **Управление компьютером**, где есть удобные средства управления пользователями. В Windows XP Home Edition или Vista Home Premium программа управления намного проще (расположена она так-

же в Панели управления), а называется она **Учетные записи пользователей** (рис. 7.1).

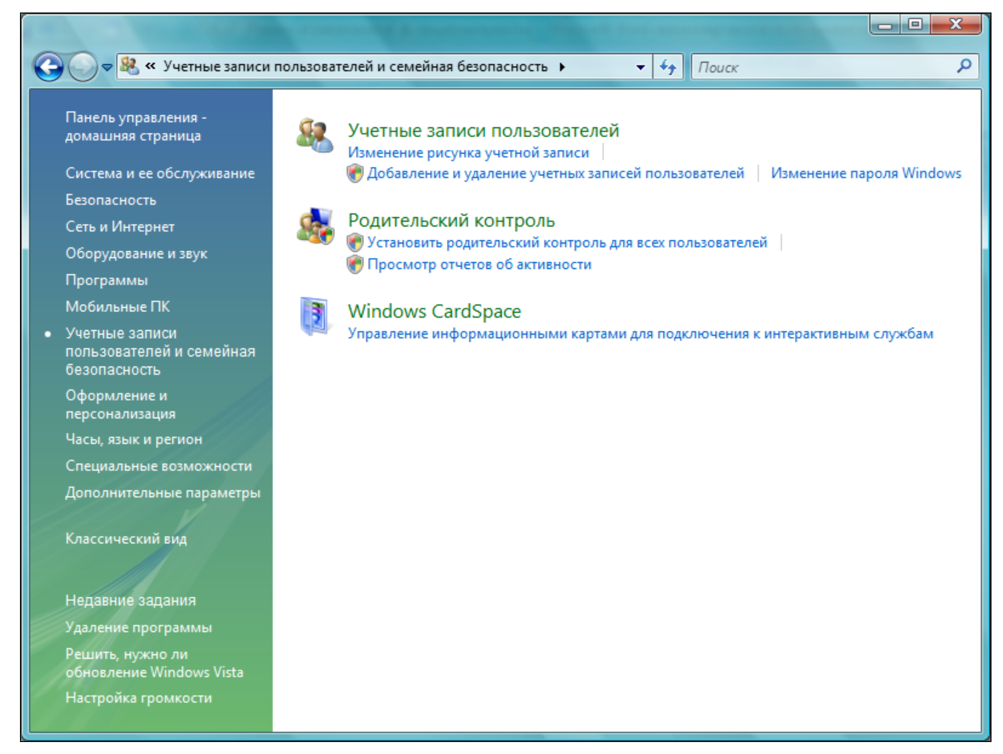


Рис. 7.1. Окно управления пользователями в Windows Vista Home Premium

Да, управлять в Windows XP Home Edition стандартными средствами практически нельзя, но сегодня мы напишем собственную утилиту, которая будет одинаково работать как в версии Professional, так и в Home Edition. Это связано с тем, что весь необходимый функционал в версии Home Edition сохранился, и никто его не вырезал кухонным ножом, ибо это кровавая операция, требующая большого количества затрат, и не столько программирования, сколько тестирования и поддержки. Поэтому все сложности, касающиеся пользователей, и их вхождения в группы просто спрятаны. Да они и не нужны владельцам упрощенной версии для дома. Например, мне дома не нужны группы и сложные права доступа, но если сильно захочется, то с помощью Netapi я смогу добиться нужного результата.

Итак, нам понадобится форма с двумя компонентами `ListBox`. В первый из них будем выводить список пользователей, а во второй — список групп, найденных на компьютере. Вполне логично, что такое приложение проще будет создать с помощью MFC Application, основанном на диалоговом окне.

В описании класса добавьте описание двух функций:

```
void GetUsersList();
void GetGroupsList();
```

Вызов обеих функций нужно написать в методе `OnInitDialog`, а код этих методов вы можете увидеть в листинге 7.1.

Листинг 7.1. Функции получения пользователей и групп

```
void CUsersListDlg::GetUsersList()
{
  LPUSER_INFO_3 pBuf = NULL;
  DWORD dwEntriesRead = 0;
  DWORD dwTotalEntries = 0;
  DWORD dwResumeHandle = 0;
  NET_API_STATUS status;

  do
  {
    status = NetUserEnum(NULL, 3, FILTER_NORMAL_ACCOUNT,
      (LPBYTE*)&pBuf, MAX_PREFERRED_LENGTH, &dwEntriesRead,
      &dwTotalEntries, &dwResumeHandle);

    if ((status != NERR_Success) && (status != ERROR_MORE_DATA))
      return;
    if (pBuf == NULL)
      return;

    for (DWORD i = 0; i< dwEntriesRead; i++)
    {
      cUsersList.AddString(pBuf[i].usri3_name);
    }
  } while (status == ERROR_MORE_DATA);

  NetApiBufferFree(pBuf);
}

void CUsersListDlg::GetGroupsList()
{
  LPGROUP_INFO_1 pBuf = NULL;
  DWORD dwEntriesRead = 0;
  DWORD dwTotalEntries = 0;
```

```
  DWORD dwResumeHandle = 0;
  NET_API_STATUS status;

  do
  {
    status = NetLocalGroupEnum(NULL, 1, (LPBYTE*)&pBuf,
    MAX_PREFERRED_LENGTH, &dwEntriesRead, &dwTotalEntries,
             &dwResumeHandle);

    if ((status != NERR_Success) && (status != ERROR_MORE_DATA))
      return;
    if (pBuf == NULL)
      return;

    for (DWORD i = 0; i< dwEntriesRead; i++)
    {
      cGroupList.AddString(pBuf[i].grpi1_name);
    }
  } while (status == ERROR_MORE_DATA);

  NetApiBufferFree(pBuf);
}
```

Сразу же покажу результат, который вы можете увидеть на рис. 7.2. Этот пример я выполнял в Windows Vista Home Premium. Запустите пример на

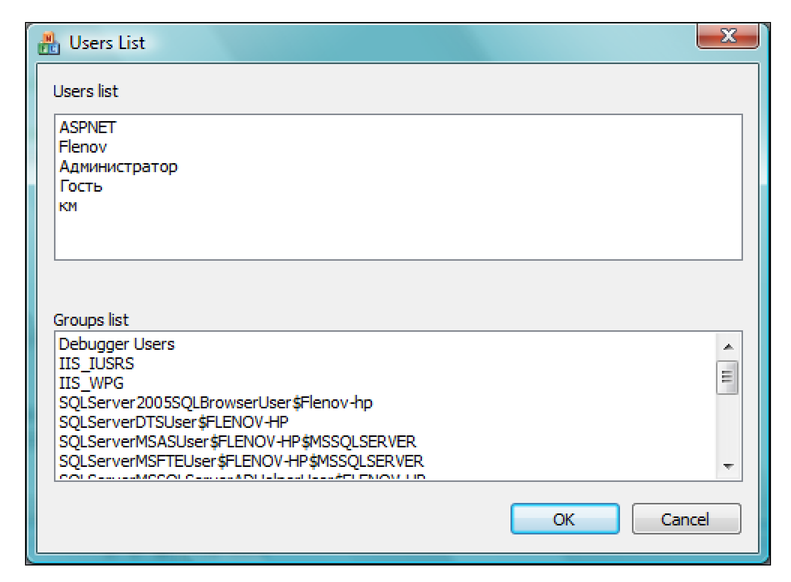


Рис. 7.2. Результат работы в Windows Vista Home Premium

своем компьютере, и если у вас версия Home, то можете заметить, что в списке намного больше пользователей, чем в Панели управления. А групп вообще таких не увидеть стандартными средствами!

Для получения списка пользователей используется функция `NetUserEnum`. В общем виде она выглядит так:

```
NET_API_STATUS NetUserEnum(
  __in          LPCWSTR servername,
  __in          DWORD level,
  __in          DWORD filter,
  __out         LPBYTE* bufptr,
  __in          DWORD prefmaxlen,
  __out         LPDWORD entriesread,
  __out         LPDWORD totalentries,
  __in_out      LPDWORD resume_handle
);
```

Тут у нас 8 параметров. Давайте рассмотрим, для чего они нужны:

- `ServerName` — строка, в которой можно указать NetBIOS- или DNS-имя компьютера, пользователей которого нужно определить. Да, если у вас есть администраторские права на чужом компьютере, то вы можете управлять и его пользователями. Если указать здесь нулевое значение, то будет использоваться локальный компьютер;
- `Level` — число, которое определяет уровень получаемой информации. В зависимости от уровня, нам будут возвращаться структуры с разной детализацией информации. Если задать 0, то в результате мы узнаем только имена пользователей. Все структуры мы рассматривать не будем, но я буду выбирать уровни, которые будут возвращать максимально полную информацию. В данном случае мы будем использовать третий уровень. Какая будет в результате информация, вы узнаете чуть позже;
- `Filter` — можно отфильтровать записи и получить пользователей только определенного типа. Вы можете указать здесь одно из следующих значений:
  - `FILTER_TEMP_DUPLICATE_ACCOUNT` — показывать все учетные записи локальных пользователей на контроллере домена;
  - `FILTER_NORMAL_ACCOUNT` — перечислить всех глобальных пользователей на компьютере;
  - `FILTER_INTERDOMAIN_TRUST_ACCOUNT` — отображать доверительные записи домена на контроллере;
- `Buffer` — это указатель на буфер, через который нам вернут результат работы;

- `PrefMaxLen` — определяет максимальную длину возвращаемых данных. Если указать константу `MAX_PREFERRED_LENGTH`, то будет выделена необходимая память для хранения всех данных. Эта константа равна –1;
- `EntriesRead` — количество реально прочитанных записей;
- `TotalEntries` — количество записей, которые могут быть прочитаны, начиная с текущего значения `ResumeHandle`. Если в `ResumeHandle` задан 0, то здесь будет общее количество записей;
- `ResumeHandle` — используется для чтения данных по частям. Укажите handle, начиная с которого нужно перечислять записи, или 0, чтобы читать с самого начала.

Тут нужно сделать еще небольшое замечание: память для буфера четвертого параметра (через который мы получаем результат) выделяется функцией, но освобождать мы должны ее сами. После работы с буфером удалите его с помощью функции `NetApiBufferFree`. Эта функция получает только один параметр — указатель на буфер.

Для разных уровней детализации существуют разные структуры данных результата. Для третьего уровня через параметр `bufptr` мы получим массив структур типа `USER_INFO_3`. Как я уже сказал, я выбрал максимально большую структуру, поэтому ради экономии места не буду ее приводить здесь (внешний вид всегда можно увидеть в MSDN), но параметры лучше рассмотреть, потому что тут много интересного. К тому же, эта структура может использоваться и в других функциях, не только при перечислении.

Итак, структура состоит из следующих полей:

- `usri3_name` — это unicode-строка, которая содержит имя пользователя. Игнорируется, когда выполняется изменение параметров пользователя, ведь при изменении нельзя менять имя;
- `usri3_password` — содержит unicode-строку с паролем. Только не стоит думать, что вы можете увидеть пароль. Это нереально. Используется только в функциях создания или изменения параметров пользователя. При чтении и перечислении здесь будет нулевое значение;
- `usri3_password_age` — если говорить прямо, то это возраст пароля, точнее, количество секунд, прошедших с момента последней смены пароля. Это параметр только для чтения, поэтому он игнорируется при добавлении или редактировании пользователя;
- `usri3_priv` — привилегии, может иметь одно из следующих значений:
  - `USER_PRIV_GUEST` — гостевая запись;
  - `USER_PRIV_USER` — непривилегированный пользователь;
  - `USER_PRIV_ADMIN` — администратор;

- usri3_home_dir — строка, определяющая путь к домашнему каталогу;
- usri3_comment — комментарий в виде unicode-строки, который связан с учетной записью пользователя;
- usri3_flags — флаги, и рассматривать все я их не буду, но основные опишу:
  - UF_ACCOUNTDISABLE — учетная запись отключена;
  - UF_PASSWD_NOTREQD — пароль не требуется;
  - UF_PASSWD_CANT_CHANGE — пользователь может менять пароль;
  - UF_DONT_EXPIRE_PASSWD — время пароля никогда не истекает;
  - UF_PASSWORD_EXPIRED — время действия пароля истекло;
- usri3_script_path — путь к файлу со сценарием, который должен выполняться во время входа в систему;
- usri3_full_name — полное имя пользователя;
- usri3_usr_comment — строка комментария;
- usri3_parms — параметры пользователя. Судя по файлу помощи, изменять этот параметр нежелательно, потому что ОС хранит здесь информацию о конфигурации;
- usri3_workstations — строка имен рабочих станций, с которых пользователь может входить в систему;
- usri3_last_logon — время последнего входа в систему;
- usri3_last_logoff — не используется;
- usri3_acct_expires — определяет дату и время, когда устареет пароль;
- usri3_max_storage — максимальный размер дискового пространства, разрешенный для использования. Если указать USER_MAXSTORAGE_UNLIMITED, то можно использовать диск неограниченного размера;
- usri3_logon_hours — строка из 168 битов (21 байт), с помощью которой можно задать время, когда пользователь может входить в систему;
- usri3_bad_pw_count — сколько раз можно указать неправильный пароль. Если 1, то количество не определено;
- usri3_num_logons — сколько раз пользователь удачно входил в систему;
- usri3_logon_server — имя сервера, которому нужно отправлять запрос на вход;
- usri3_country_code и usri3_code_page — код страны и кодовая страница;

- usri3_user_id — идентификатор пользователя;
- usri3_primary_group_id — идентификатор первичной группы;
- usri3_profile — путь к пользовательскому профилю;
- usri3_home_dir_drive — строка, содержащая букву диска, с пользовательским каталогом.

Итак, следующая строка возвращает нам список всех пользователей в системе:

```
status = NetUserEnum(NULL, 3, FILTER_NORMAL_ACCOUNT,
   (LPBYTE*)&pBuf, MAX_PREFERRED_LENGTH, &dwEntriesRead,
    &dwTotalEntries, &dwResumeHandle);
```

После выполнения функции NetUserEnum в переменной lpBuffer будет находиться указатель на память, где расположен массив из структур PUserInfo3. Количество записей в этом массиве можно определить по содержимому переменной dwEntiesRead. Если все прошло удачно, то функция вернет нам NERR_Success или ERROR_MORE_DATA. Первая говорит о том, что все прошло успешно, а вторая сообщает о том, что еще есть данные.

Теперь поговорим о получении списка групп. Принцип работы аналогичен, только необходимо использовать функцию NetLocalGroupEnum:

```
NET_API_STATUS NetLocalGroupEnum(
  __in          LPCWSTR servername,
  __in          DWORD level,
  __out         LPBYTE* bufptr,
  __in          DWORD prefmaxlen,
  __out         LPDWORD entriesread,
  __out         LPDWORD totalentries,
  __in_out      PDWORD_PTR resumehandle
);
```

Даже без увеличительного стекла видно, что эта функция очень сильно похожа на NetUserEnum. По крайней мере, параметры имеют тот же смысл. Разница только во втором параметре и третьем, и то эта разница чисто идеологическая. Дело в том, что во втором параметре мы также передаем уровень детализации информации, просто эти уровни немного другие, и количество вариантов отличается.

Итак, для групп существует всего два уровня детализации. На нулевом нам вернут массив структур, в которых будет только название группы, а на первом уровне нам вернут и название группы, и комментарий. Мы будем использовать уровень 1, поэтому в качестве третьего параметра получим массив из структур GROUP_INFO1, а эта структура выглядит так:

```
typedef struct _LOCALGROUP_INFO_1 {
  LPWSTR lgrpi1_name;
  LPWSTR lgrpi1_comment;
}
```

Как видите, у структуры всего два члена:

- `grpi1_name` — название группы;
- `grpi1_comment` — комментарий.

Чтобы превратить эту структуру в формат, соответствующий уровню 0, достаточно удалить параметр `grpi1_comment`.

В примере, который мы рассмотрели в этой главе, я вывожу в форме только имена пользователей и групп. При этом код перечисления пользователей получает максимальную информацию, и вы можете отображать любые другие данные из описанных структур.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter7\UserList.

7.1.2. Управление пользователями

Давайте посмотрим, какие еще функции нам доступны из библиотеки NetApi32. Чтобы получить информацию о конкретном пользователе, можно использовать функцию перечисления, как было показано в *разд. 7.1.1*, но лучше воспользоваться функцией `NetUserGetInfo`. Эта функция возвращает информацию только об определенном пользователе, что намного экономнее и эффективнее.

В общем виде функция `NetUserGetInfo` выглядит следующим образом:

```
NET_API_STATUS NetUserGetInfo(
  __in          LPCWSTR servername,
  __in          LPCWSTR username,
  __in          DWORD level,
  __out         LPBYTE* bufptr
);
```

Здесь у нас всего четыре параметра:

- `servername` — имя компьютера, учетную запись пользователя которого нужно прочитать. Если указать здесь ноль, то система будет искать пользователя на локальном компьютере;
- `username` — имя пользователя, данные которого нужно вернуть;

- level — уровень детализации информации. Здесь можно указывать те же номера уровней, что и для функции NetUserEnum;
- bufptr — указатель, по адресу которого будет возвращена структура с данными о пользователе. Вид структуры зависит от запрошенного уровня, и если задать в параметре level число 3, то указатель данного параметра будет ссылаться на структуру типа PUserInfo3.

Получив структуру с данными о пользователях, можно отобразить информацию в каком-либо окне для редактирования. После редактирования необходимо сохранить эту информацию в системе. Для этого можно использовать функцию NetUserSetInfo:

```
NET_API_STATUS NetUserSetInfo(
  __in          LPCWSTR servername,
  __in          LPCWSTR username,
  __in          DWORD level,
  __in          LPBYTE buf,
  __out         LPDWORD parm_err
);
```

Функция схожа с NetUserGetInfo, только добавлен один параметр parm_err, в котором в случае ошибки будет находиться индекс параметра структуры пользовательских данных (PUserInfo), который привел к ошибке. Эти данные носят информационный характер, и их лучше не изменять.

Теперь перейдем к добавлению нового пользователя. Для этого нехитрого действия используется функция NetUserAdd, которая выглядит следующим образом:

```
NET_API_STATUS NetUserAdd(
  __in          LMSTR servername,
  __in          DWORD level,
  __in          LPBYTE buf,
  __out         LPDWORD parm_err
);
```

По именам параметров можно уже догадаться, для чего они нужны, потому что подобные параметры мы уже видели. Но, на всякий случай, коротко по ним пробежимся:

- servername — имя компьютера;
- level — уровень, в зависимости от которого система определяет, какую структуру мы ей передали и сколько информации о пользователе предоставили;

- buf — указатель на структуру PUserInfo. Для третьего уровня здесь нужно указать структуру PUserInfo3, которую мы рассматривали в *разд. 7.1.1*;
- parm_err — в случае ошибки нам вернут индекс поля в структуре данных о пользователе.

Для удаления пользователя используется функция NetUserDel, которая выглядит следующим образом:

```
NET_API_STATUS NetUserDel(
  __in          LPCWSTR servername,
  __in          LPCWSTR username
);
```

Тут всего два параметра: имя сервера и имя пользователя, которого нужно удалить на этом сервере.

Для управления группами существуют подобные функции, с похожими именами, но только влияют они на группы:

- NetLocalGroupGetInfo — возвращает информацию о локальной группе;
- NetLocalGroupSetInfo — назначает новые параметры указанной группе;
- NetLocalGroupAdd — добавляет новую группу;
- NetLocalGroupDel — удаляет группу.

На этом мы пока завершим рассмотрение функций Network Management SDK. Чтобы рассмотреть все возможности этого пакета разработчика, понадобится отдельная книга.

Готовых примеров по описанным здесь функциям не будет. Можете воспринимать это как домашнее задание. Попробуйте сами разобраться и написать программу, которая будет добавлять пользователя, изменять его данные и удалять. Все необходимые знания у вас уже есть.

7.2. Права доступа к объектам

Очень часто, создавая объекты программно, мы не задумываемся о безопасности, оставляя ее параметры на усмотрение ОС. Но ведь управление правами доступа — не такая уж и сложная операция, она требует всего нескольких лишних строчек кода и немного понимания работы соответствующих функций. Знание функций управления доступом может пригодиться не только при создании объектов.

В этом разделе мы познакомимся со средствами контроля доступа ОС Windows, с SID (Security ID, идентификатор безопасности), Security Attributes (атрибуты безопасности), Security Descriptor (дескриптор безопасности) и всем, что связано с этими понятиями.

7.2.1. Дескриптор безопасности

Начнем наше знакомство с дескриптора безопасности. В качестве примера рассмотрим создание файла/каталога. У обеих функций (`CreateFile` и `CreateDirectory`) есть одинаковый параметр — указатель на структуру `SECURITY_ATTRIBUTES`. У функции `CreateFile` этот указатель четвертый по счету, а у `CreateDirectory` он второй. Как я уже сказал, в большинстве случаев его просто игнорируют. Но давайте рассмотрим детальнее, что это за структура. Она определяет атрибуты безопасности и состоит всего из трех полей:

```
typedef struct _SECURITY_ATTRIBUTES
{
 DWORD nLength;
 LPVOID lpSecurityDescriptor;
 BOOL bInheritHandle;
} SECURITY_ATTRIBUTES,
  *PSECURITY_ATTRIBUTES,
  *LPSECURITY_ATTRIBUTES;
```

Первое поле определяет размер структуры. Подобные поля можно встретить в большинстве WinAPI-структур. Второе поле — указатель на дескриптор безопасности. Третий параметр — булево значение, которое определяет, может ли полученный указатель наследоваться дочерними процессами. Наследование дескрипторов нас пока не интересует, поэтому в примере, который будем рассматривать, установим это значение в `false`.

Самое интересное — это второй параметр. Это указатель на дескриптор безопасности, который на самом деле ничего страшного собой не представляет.

Для каждого объекта ОС создает дескриптор безопасности, по которому определяются права доступа к объекту, его владелец, группа, а также списки SACL (System Access Control List, системный список контроля доступа) и DACL (Discretionary Access Control List, список разграничительного контроля доступа). Мы рассматриваем программирование, поэтому сделаем упор на рассмотрение самих функций работы со списками доступа. Благодаря практическим примерам можно будет понять, для чего нужны эти списки и где они используются, а пока не будем останавливаться, а двинемся дальше.

Итак, давайте посмотрим, что такое дескриптор с точки зрения программирования. На самом деле это структура, которая имеет следующий вид:

```
typedef struct _SECURITY_DESCRIPTOR {
   BYTE  Revision;
   BYTE  Sbz1;
   SECURITY_DESCRIPTOR_CONTROL Control;
```

```
   PSID Owner;
   PSID Group;
   PACL Sacl;
   PACL Dacl;
} SECURITY_DESCRIPTOR, *PISECURITY_DESCRIPTOR;
```

В файле помощи эта структура описана только общими словами. Из чего она состоит, можно определить только по заголовочному файлу winnt.h:

- `Revision` — ревизия. Этот параметр должен быть равен единице, а лучше использовать константу `SECURITY_DESCRIPTOR_REVISION`. В некоторых источниках утверждается, что можно указывать константу `SECURITY_DESCRIPTOR_REVISION1` (мол, она предоставляет доступ к новым возможностям). Заверяю, что обе константы в файле winnt.h равны 1 и не могут предоставлять нам различные возможности;
- `Sbz1` — не используется и должен быть равен нулю. Возможно, это поле служит для выравнивания;
- `Control` — несмотря на такой интересный имидж, это поле имеет тип данных Word и содержит флаги;
- `Owner` — идентификатор безопасности SID владельца;
- `Group` — идентификатор безопасности SID группы;
- `Sacl` — указатель на SACL;
- `Dacl` — указатель на DACL.

Итак, данная структура назначается какому-то объекту в системе (файлу, процессу), и система определяет (с помощью полей `Owner` и `Group`), кто является владельцем связанного с данной структурой объекта, а на основе списков `Sacl` и `Dacl` — кто имеет права доступа к объекту.

Последние два параметра наиболее интересны с точки зрения безопасности, но потерпите еще чуть-чуть! Скоро мы их подробненько рассмотрим.

Несмотря на то, что дескриптор безопасности легко описать в виде структуры, работать с ним напрямую не рекомендуется. Наверно, поэтому его не описывают в файле справки. Дело в том, что дескриптор может хранить данные непосредственно, а может просто содержать указатель на данные, которые могут находиться совершенно в другом месте в памяти.

Вместо прямого доступа необходимо использовать специализированные функции. Мы остановимся на трех из них: инициализация, установка владельца объекта и установка группы. Да, данной структуре необходима инициализация, ведь она может хранить указатели на данные, а любой указатель требует выделения памяти.

Для инициализации дескриптора безопасности используем WinAPI-функцию `InitializeSecurityDescriptor`, которая выглядит следующим образом:

```
BOOL WINAPI InitializeSecurityDescriptor(
  __out          PSECURITY_DESCRIPTOR pSecurityDescriptor,
  __in           DWORD dwRevision
);
```

Тут у нас два параметра: указатель на дескриптор безопасности, который нужно инициализировать, и номер ревизии. Как мы уже выяснили, ревизия должна быть равна константе `SECURITY_DESCRIPTOR_REVISION`.

Чтобы установить владельца объекта, используется функция `SetSecurityDescriptorOwner`, которая выглядит следующим образом:

```
BOOL WINAPI SetSecurityDescriptorOwner(
  __in_out       PSECURITY_DESCRIPTOR pSecurityDescriptor,
  __in_opt       PSID pOwner,
  __in           BOOL bOwnerDefaulted
);
```

Тут три параметра:

- дескриптор, владельца объекта которого нужно изменить;
- указатель на SID пользователя, которого мы хотим установить в качестве владельца;
- нужно ли использовать владельца по умолчанию (`true` — владельца назначит ОС в соответствии со своими правилами. А правило простое — создатель становится и владельцем).

Для установки группы используем функцию `SetSecurityDescriptorGroup`, которая выглядит так:

```
BOOL WINAPI SetSecurityDescriptorGroup(
  __in_out       PSECURITY_DESCRIPTOR pSecurityDescriptor,
  __in_opt       PSID pGroup,
  __in           BOOL bGroupDefaulted
);
```

Функция очень похожа на установку владельца. Тут у нас опять три параметра:

- дескриптор, группу объекта которого нужно изменить;
- указатель на SID группы, которую мы хотим установить в качестве владельца;
- нужно ли использовать группу по умолчанию, т. е. понадеяться на ОС.

Списки доступа SACL и DACL пока не будем устанавливать. Отложим эту тему ненадолго. Сейчас нас интересуют владелец и группа, но чтобы их установить, необходимо знать соответствующий SID. Да, мы всегда можем использовать значение по умолчанию, которое предоставляет ОС, но определить SID не так уж и сложно. Для этого нужна функция `LookupAccountName`, которая по имени пользователя возвращает идентификатор безопасности SID.

В общем виде функция выглядит следующим образом:

```
BOOL WINAPI LookupAccountName(
  __in_opt      LPCTSTR lpSystemName,
  __in          LPCTSTR lpAccountName,
  __out_opt     PSID Sid,
  __in_out      LPDWORD cbSid,
  __out_opt     LPTSTR ReferencedDomainName,
  __in_out      LPDWORD cchReferencedDomainName,
  __out         PSID_NAME_USE peUse
);
```

Давайте рассмотрим параметры этой функции:

- `lpSystemName` — имя системы. Если этот параметр нулевой, то мы ищем локального пользователя, если нужен SID удаленного пользователя, то необходимо указать здесь имя этой машины;
- `lpAccountName` — имя пользователя, идентификатор которого нам нужен;
- `Sid` — указатель на память, куда будет записан результат;
- `cbSid` — длина буфера, которую мы выделили для параметра `Sid`, т. е. для хранения результирующего идентификатора;
- `ReferencedDomainName` — имя домена;
- `cbReferencedDomainName` — длина буфера `ReferencedDomainName`;
- `peUse` — переменная типа `enum`, которая определяет тип учетной записи. Здесь может быть одно из следующих значений:
  - `SidTypeUser` — пользовательский SID;
  - `SidTypeGroup` — SID группы;
  - `SidTypeDomain` — SID доменной учетной записи;
  - `SidTypeAlias` — псевдоним;
  - `SidTypeDeletedAccount` — удаленная учетная запись;
  - `SidTypeInvalid` — тип некорректный;
  - `SidTypeUnknown` — тип не известен;
  - `SidTypeComputer` — идентификатор компьютера.

Что такое SID, хорошо иллюстрирует заголовочный файл winnt.h. Запускаем поиск по трем магическим буквам — SID — и натыкаемся на табличку (рис. 7.3), которую можно увидеть где-то рядом с этим текстом.

Нетрудно догадаться, что на самом деле SID — это структура следующего вида:

- `SubAuthorityCount` — количество записей SubAuthority;
- `Revision` — версия, в ней используются только четыре бита, остальные зарезервированы;
- `IdentifierAuthority` — структура, которая хранит идентификатор SID;
- `SubAuthority` — массив относительных идентификаторов.

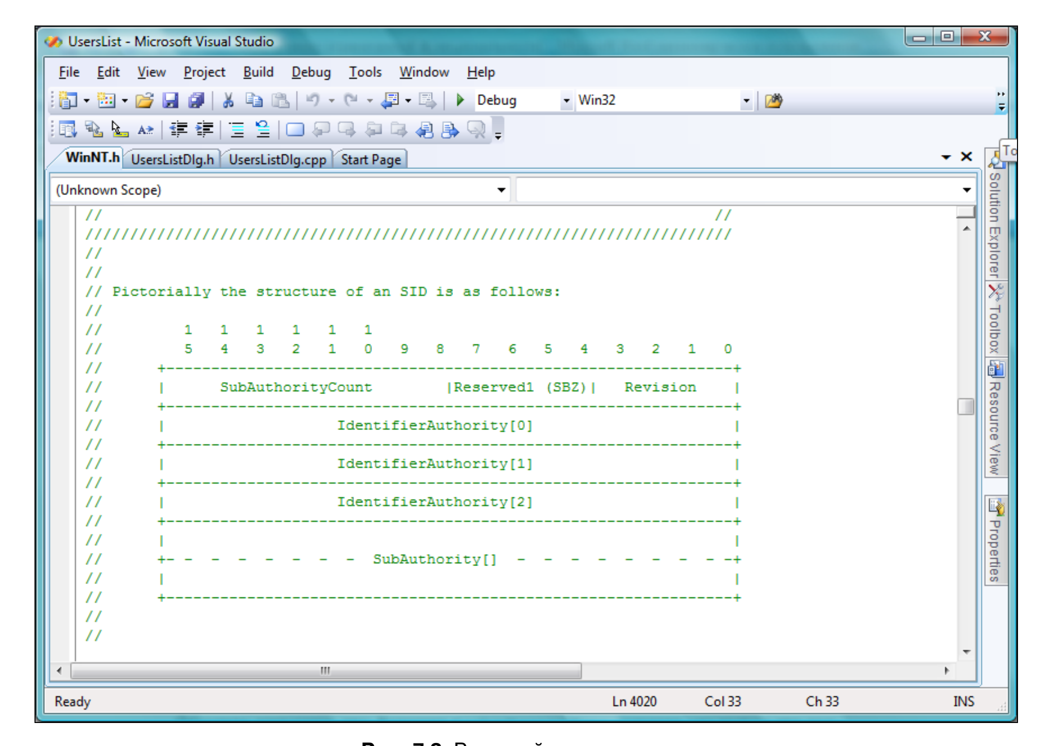


Рис. 7.3. Внешний вид структуры

Структура `IdentifierAuthority`, которую вы можете видеть во втором параметре идентификатора SID, имеет вид:

```
typedef struct _SID_IDENTIFIER_AUTHORITY {
    BYTE  Value[6];
} SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;
```

Банальный массив из шести байтов! Прямая работа с SID не рекомендуется. Для манипулирования этой структурой в WinAPI есть все необходимые функции, и лучше использовать их. Но это уже отдельная история.

Обратите внимание, что функции `LookupAccountName`, которую мы только что рассмотрели, необходимо передать указатель на память для хранения идентификатора SID и указать размер. Проблема в том, что нет четко определенного размера идентификатора. Сколько же тогда памяти выделять для хранения результата?

Чтобы определить, достаточно вызвать функцию `LookupAccountName`, указав в качестве указателя на буфер для хранения SID и в качестве размера буфера — нулевое значение. В результате функция вернет ошибку и сообщит, что недостаточно памяти в буфере. А через параметры `cbSid` и `cbReferencedDomainName` вернет нам корректные размеры необходимых буферов. Теперь мы знаем все необходимое.

Для иллюстрации примера работы с дескрипторами давайте напишем программу (листинг 7.2). Так как интерфейса с пользователем не будет, я решил не создавать окна, а ограничиться консольным приложением.

Листинг 7.2. Код создания файла

```
// CreateSecFile.cpp : Defines the entry point for the
// console application.

#include "stdafx.h"
#include "windows.h"

int _tmain(int argc, _TCHAR* argv[])
{
  DWORD sidLength = 0;
  DWORD sidLengthDomain = 0;
  SID_NAME_USE sidType;
  LookupAccountName(NULL, L"Flenov", NULL, &sidLength, NULL,
               &sidLengthDomain, &sidType);

  PSID sidValue = (PSID)new BYTE[sidLength];
  LPTSTR wszDomainName = new WCHAR[sidLengthDomain];

  if (!LookupAccountName(NULL, L"Flenov", sidValue, &sidLength,
           wszDomainName, &sidLengthDomain, &sidType))
  {
    printf("Account do not found");
```

```
    return 0;
  }

  printf("User Flenov has found\n");

  SECURITY_DESCRIPTOR securityDescriptor;
  InitializeSecurityDescriptor(&securityDescriptor,
            SECURITY_DESCRIPTOR_REVISION);
  SetSecurityDescriptorOwner(&securityDescriptor, sidValue, true);
  SetSecurityDescriptorGroup(&securityDescriptor, NULL, true);

  SECURITY_ATTRIBUTES securityAttributes;
  securityAttributes.nLength = sizeof(SECURITY_ATTRIBUTES);
  securityAttributes.lpSecurityDescriptor = &securityDescriptor;
  securityAttributes.bInheritHandle = false;

  HANDLE h = CreateFile(L"c:\\Temp\\test.txt", GENERIC_ALL, 0,
    &securityAttributes, CREATE_ALWAYS, FILE_FLAG_BACKUP_SEMANTICS, 0);

  if (h == INVALID_HANDLE_VALUE)
  {
    printf("Can't create file");
    return 0;
  }

  printf("File has just creared\n");
  CloseHandle(h);

  return 0;
}
```

Тут нужно иметь в виду, что при создании файла ваша учетная запись должна находиться в списке. Нельзя поместить в список только одну учетную запись пользователя, да еще не активную в данный момент. Опыт показывает, что создатель файла должен быть в списке и иметь полные права. Именно их мы можем настроить. Мы можем в списках ACL описать не только себя, но и права доступа других пользователей системы. Например, уже на этапе создания файла можно запретить доступ со стороны Everybody.

Примечание

Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter7\CreateSecFile.

7.2.2. Дескриптор безопасности

Продолжаем знакомиться с функциями определения прав доступа к объектам, и на этот раз нам предстоит научиться определять, кому и какие права доступны в отношении определенного объекта. Тема будет интересна не только программистам, потому что, прочитав статью, вы лучше поймете, как Windows хранит списки доступа к объектам и как они организованы.

Права доступа на объекты ОС хранятся в списке DACL. Данный список можно получить к любому объекту, который защищен ОС, а в Windows не защищены разве что только элементы управления в окнах. Сами окна, сервисы, программы и тем более файлы имеют DACL, и по его содержимому можно определить, кто и что может делать с объектом.

Пока не будем заострять внимание на том, что представляет собой этот список: если вы не знаете, то скоро все встанет на свои места. Давайте пока научимся получать сам DACL. Для этого используется функция `GetNamedSecurityInfo`. В общем виде она выглядит следующим образом:

```
DWORD WINAPI GetNamedSecurityInfo(
  __in          LPTSTR pObjectName,
  __in          SE_OBJECT_TYPE ObjectType,
  __in          SECURITY_INFORMATION SecurityInfo,
  __out_opt     PSID* ppsidOwner,
  __out_opt     PSID* ppsidGroup,
  __out_opt     PACL* ppDacl,
  __out_opt     PACL* ppSacl,
  __out_opt     PSECURITY_DESCRIPTOR* ppSecurityDescriptor
);
```

Рассмотрим параметры этой функции, тут есть над чем пораскинуть мозгами:

- `PObjectName` — имя объекта, список доступа которого мы хотим получить. Если это файл, то необходимо указать корректный путь, чтобы программа смогла найти его. Если это имя сервиса или принтера, то имя должно выглядеть так: `\\имя_компьютера\имя_объекта`, где `имя_объекта` — это имя сервиса или принтера;
- `ObjectType` — определяет тип объекта. Здесь можно указать одну из следующих констант (можно лишний раз понять, что именно может защищать ОС Windows с помощью дескрипторов):
  - `SE_FILE_OBJECT` — файл или каталог;
  - `SE_SERVICE` — сервис;
  - `SE_PRINTER` — принтер;
  - `SE_REGISTRY_KEY` — ключ реестра;

- SE_LMSHARE — совместно используемый ресурс;
- SE_KERNEL_OBJECT — объект ядра (процесс, поток, семафор, событие);
- SE_WINDOW_OBJECT — окно или объект рабочего стола;

❒ SecurityInfo — комбинация флагов, по которой определяется, что именно мы хотим узнать:

- OWNER_SECURITY_INFORMATION — идентификатор владельца. Если указан этот флаг, то результат будет получен через переменную ppsidOwner;
- GROUP_SECURITY_INFORMATION — идентификатор группы. Если указан этот флаг, то результат будет получен через переменную ppsidGroup;
- DACL_SECURITY_INFORMATION — список DACL. Если указан этот флаг, то результат будет получен через переменную ppDacl;
- SACL_SECURITY_INFORMATION — список SACL. Если указан этот флаг, то результат будет получен через переменную ppSacl;

❒ ppSecurityDescriptor — через этот параметр нам будет возвращен дескриптор безопасности.

Итак, чтобы получить DACL, мы должны написать следующую строку кода:

```
if (GetNamedSecurityInfo(L"C:\\\\Temp\\test.txt", SE_FILE_OBJECT,
   DACL_SECURITY_INFORMATION, NULL, NULL, &pDACL, NULL,
   &pSD) != ERROR_SUCCESS)
{
  printf("Get security information error");
  return 0;
}
```

В этом примере переменные pSD и pDACL должны быть объявлены следующим образом:

```
PACL pDACL = NULL;
PSECURITY_DESCRIPTOR pSD = NULL;
```

Структуру типа ACL мы получили, но это еще не собственно информация. Сам список доступа можно получить с помощью функции GetAclInformation, которая описана следующим образом:

```
BOOL WINAPI GetAclInformation(
  __in          PACL pAcl,
  __out         LPVOID pAclInformation,
  __in          DWORD nAclInformationLength,
  __in          ACL_INFORMATION_CLASS dwAclInformationClass
);
```

Посмотрим на параметры этой функции:

- pAcl — указатель на структуру типа TACL, которую мы получили при вызове функции GetNamedSecurityInfo;
- pAclInformation — указатель, по которому мы получим результирующую информацию;
- nAclInformationLength — размер параметра, указанного в pAclInformation;
- dwAclInformationClass — класс необходимой информации. Нас интересует класс AclSizeInformation.

Самое интересное — это второй параметр, он должен указывать на структуру типа ACL_SIZE_INFORMATION.

Итак, получить информацию можно, выполнив следующую строку:

```
if (!GetAclInformation(pDACL, &aclInfo, sizeof(aclInfo),
    AclSizeInformation))
{
  printf("Can't get ACL info");
  return 0;
}
```

Первую переменную мы уже объявили, а aclInfo нужно объявить как:

```
ACL_SIZE_INFORMATION   aclInfo;
```

У нас есть список, и теперь осталось только его просмотреть и определить права доступа. Список DACL состоит из набора структур ACE (Access Control Entry, запись контроля доступа) и идентификаторов SID. В файле помощи перечислено всего четыре типа списков. Забегая вперед, скажу, что их намного больше. Другое дело, что не все они поддерживаются, да и нужны нам только два типа — разрешение (ACCESS_ALLOWED_ACE) и запрещение (ACCESS_DENIED_ACE). Как они выглядят?

Структура ACE_HEADER должна выглядеть примерно так:

```
typedef struct _ACE_HEADER {
  UCHAR  AceType;
  UCHAR  AceFlags;
  USHORT  AceSize;
} ACE_HEADER;
typedef ACE_HEADER *PACE_HEADER;
```

Понятно, что назвать ее можно как угодно, и поля тоже можно назвать как душа пожелает, главное — количество и типы параметров именно такие.

Первый параметр структуры — это тип записи. Нас будут интересовать только ACCESS_ALLOWED_ACE и ACCESS_DENIED_ACE_TYPE.

С заголовком определились, но что же будет идти после заголовка в записи ACE? Опять приходится обращаться к MSDN, где мы узнаем, что записи разрешения и запрещения выглядят абсолютно одинаково. В обоих случаях после заголовка идет два значения: маска доступа типа ACCESS_MASK и число типа DWORD, которое определяет идентификатор SID пользователя или группы, в отношении которого это разрешение действует.

Так как разрешающая и запрещающая запись ACE выглядят одинаково, то мы можем объявить одну структуру следующего вида:

```
ACCESS_ACE = record
  Header : _ACE_HEADER;
  Mask : ACCESS_MASK;
  SidStart : DWORD;
end;
```

Возвращаемся к практике. В переменной aclInfo у нас уже есть информация о записях ACL. Теперь нужно просто просмотреть его и вывести информацию на экран. Для этого цикл должен выглядеть примерно следующим образом:

```
for (DWORD i = 0; i < aclInfo.AceCount; i++)
{
  void *ace;
  if (GetAce(pDACL, i, &ace))
  {
  }
}
```

Список ACL содержит только указатели на ACE-записи. Напоминаю, что ACL — это список контроля доступа, а ACE — это отдельная запись этого списка. Чтобы получить запись из списка контроля доступа, нужно использовать функцию GetAce. Ей передается три параметра:

- указатель на ACL;
- индекс интересующей нас записи;
- указатель на переменную, в которую будет записан результат.

Если результат ненулевой, значит, процесс получения записи прошел удачно. В предыдущем примере в качестве третьего параметра передается указатель переменной ACE. Ее нужно объявить следующим образом:

```
void *ace;
```

Это указатель на неопределенные данные, потому что мы уже знаем, что может быть два типа разрешений (разрешено или запрещено). Да, оба варианта описываются одинаковыми структурами, но все же в будущих версиях ОС могут быт различия, поэтому давайте хоть немного придерживаться хорошего тона.

Просматривая список, мы получили ACE-запись и по ней теперь должны определить, что разрешено, а что запрещено. Но одна запись может быть либо разрешающая, либо запрещающая. Нет такой записи, которая описывала и то, и другое сразу. Если нужно что-то запретить, а что-то разрешить, то в списке ACL создаются две записи ACE. Одна запись что-то разрешает, а другая что-то запрещает.

Чтобы определить, какая именно перед нами запись, необходимо проверить поле `AceType` заголовка ACE-записи:

```
if (((ACCESS_ALLOWED_ACE *) ace)->Header.AceType ==
                             ACCESS_ALLOWED_ACE_TYPE)
   printf(" Allow ");
else
   printf(" Deny ");
```

Константы `ACCESS_ALLOWED_ACE_TYPE` и `ACCESS_DENIED_ACE_TYPE` объявлены в заголовочном файле равными нулю и единице соответственно.

Теперь, если это разрешение или запрещение, нужно узнать, что именно разрешается. Об этом можно узнать из параметра `mask`. Его основные биты (первые три):

- чтение;
- запись;
- выполнение.

А на какого пользователя или группу влияют данные разрешения? Об этом можно узнать по полю `SidStart` структуры ACE. Я говорил, что переменная `ace` указывает на неопределенные данные, так вот, для удобства нужно произвести следующее приведение типов:

```
SID *SidStart=(SID *) &((ACCESS_ALLOWED_ACE *) ace)->SidStart;
```

Теперь по идентификатору необходимо узнать имя пользователя и домен, в котором он зарегистрирован. Для этого используем функцию `LookupAccountSid`, которая выглядит так:

```
BOOL WINAPI LookupAccountSid(
  __in_opt      LPCTSTR lpSystemName,
  __in          PSID lpSid,
```

```
  __out_opt     LPTSTR lpName,
  __in_out      LPDWORD cchName,
  __out_opt     LPTSTR lpReferencedDomainName,
  __in_out      LPDWORD cchReferencedDomainName,
  __out         PSID_NAME_USE peUse
);
```

Быстренько пробежимся по параметрам этой функции:

- `lpSystemName` — указатель на строку для системного имени. Мы будем определять пользователя по SID, поэтому этот параметр должен быть нулевой;
- `Sid` — идентификатор интересующего нас пользователя;
- `Name` — указатель на строку, куда будет записано имя пользователя;
- `cbName` — размер строки для имени пользователя;
- `ReferencedDomainName` — строка для имени домена;
- `cbReferencedDomainName` — размер строки с именем домена;
- `peUse` — структура, определяющая тип записи.

Пример использования функции:

```
LPWSTR user = new WCHAR[200];
LPWSTR domain = new WCHAR[200];

if (LookupAccountSid(NULL, SidStart, user, &len,
        domain, &len, &sid_nu))
{
   wprintf(L"Account: %ws ", user);
}

delete [] user;
delete [] domain;
```

Полный код примера, определяющего права доступа на файл, можно увидеть в листинге 7.3.

Листинг 7.3. Определение прав доступа на файл

```
// SecDesc.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Aclapi.h"
```

```
int _tmain(int argc, _TCHAR* argv[])
{
  PACL pDACL = NULL;
  PSECURITY_DESCRIPTOR pSD = NULL;

  // Получить информацию о безопасности
  if (GetNamedSecurityInfo(L"C:\\\\Temp\\test.txt", SE_FILE_OBJECT,
          DACL_SECURITY_INFORMATION, NULL, NULL, &pDACL, NULL,
          &pSD) != ERROR_SUCCESS)
  {
    printf("Get security information error");
    return 0;
  }

  if (pDACL == NULL)
  {
    printf("ACL list is empty");
    return 0;
  }

  // получить ACL-информацию
  ACL_SIZE_INFORMATION aclInfo;
  if (!GetAclInformation(pDACL, &aclInfo, sizeof(aclInfo),
          AclSizeInformation))
  {
    printf("Can't get ACL info");
    return 0;
  }

  SID_NAME_USE sid_nu;
  DWORD len = 200;
  // Цикл перебора всех ACL-записей
  for (DWORD i = 0; i < aclInfo.AceCount; i++)
  {
    LPWSTR user = new WCHAR[200];
    LPWSTR domain = new WCHAR[200];
    void *ace;

    // Получить текущую запись
    if (GetAce(pDACL, i, &ace))
    {
      SID *SidStart = (SID *) &((ACCESS_ALLOWED_ACE *) ace)->SidStart;
```

```
      if (LookupAccountSid(NULL, SidStart, user, &len,
                domain, &len, &sid_nu))
      {
        wprintf(L"Account: %ws ", user);
      }
      if (((ACCESS_ALLOWED_ACE *) ace)->Header.AceType ==
                           ACCESS_ALLOWED_ACE_TYPE)
         printf(" Allow ");
      else
         printf(" Deny ");

      if (((ACCESS_ALLOWED_ACE *) ace)->Mask & 1)
         printf(" Read ");
      if (((ACCESS_ALLOWED_ACE *) ace)->Mask & 2)
         printf(" Write ");
      if (((ACCESS_ALLOWED_ACE *) ace)->Mask & 4)
         printf(" Execute ");
      printf(" \n");
    }
    delete [] user;
    delete [] domain;
  }

  return 0;
}
```

Главное, что нужно не забыть при написании примера, — это подключить заголовочный файл Aclapi.h. Никаких библиотек подключать не нужно, достаточно только заголовочного файла.

Примечание

Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter7\SecDesc.

7.2.3. Изменение дескриптора безопасности

Мы уже узнали, как можно определить права доступа на определенный файл в файловой системе NTFS. В этот раз нам предстоит пойти дальше, и мы научимся модифицировать таблицу, добавляя или удаляя определенные права доступа. Все будет происходить программно, и никакого колдовства! Только ловкость рук и немного смекалки. Не забываем, что права доступа могут быть не только у файлов, но и у других объектов ОС.

Давайте, для начала, разберемся, как нужно устанавливать права на объект ОС, а потом познакомимся с функциями и практическим примером. Под объектом в данном случае понимаются процессы и файлы, находящиеся в NTFS. Напоминаю, что в FAT32 слишком скудные возможности по защите файлов, и здесь списки ACL не работают, их просто невозможно сохранить на диске, ведь информация о доступе хранится в файловой системе, а в FAT32 эту информацию негде хранить.

Если вы все еще используете FAT32, то рекомендую сегодня же конвертировать его в NTFS, и ничего не бойтесь! Система NTFS намного надежнее и лучше. На рис. 7.4 можно увидеть окно свойств файла, а точнее — его вкладку **Security** (Безопасность), где можно настраивать доступ.

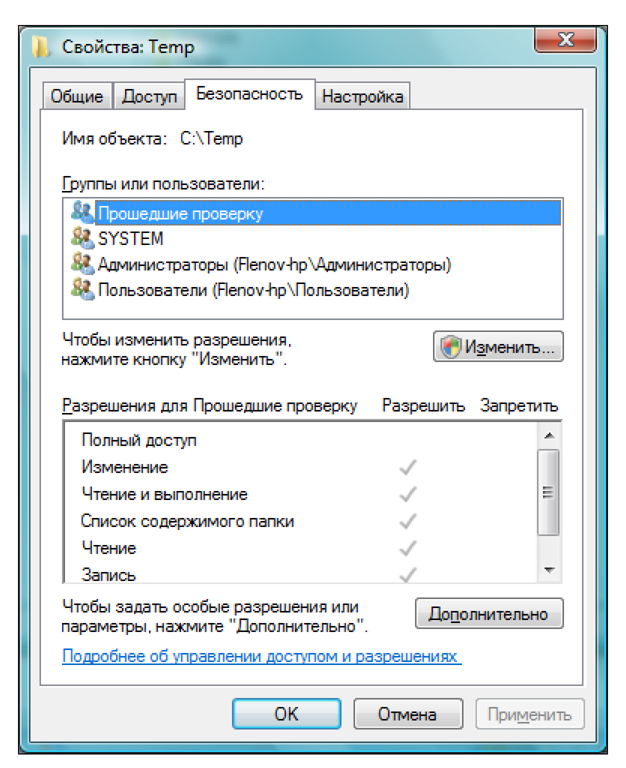


Рис. 7.4. Права доступа в NTFS

У каждого объекта есть свой дескриптор. В нем содержится указатель на список ACL, а уже в этом списке можно найти записи, которые определяют права доступа. Самый простой способ удалить определенную запись из списка — скопировать все необходимые записи в новый список и установить их дескриптору, а затем дескриптор назначить файлу. Так как списки в основном небольшие и занимают мало места, такой фокус пролетит мгновенно.

Чтобы добавить разрешение или запрет, необходимо расширить память для списка ACL и втиснуть туда новую запись. Можно поступить проще: опять же, выделить память для нового списка — хранить все старые записи и одну новую. Теперь копируем в новый ACL все старые записи и добавляем новую. Остается только назначить созданный список дескриптору, а затем и файлу.

Для написания примера нам понадобятся все знания, которые мы получили в двух предыдущих разделах этой главы, плюс маленькая корзина новых функций. Давайте напишем пример, который будет добавлять в ACL новую разрешающую запись для всех пользователей на определенный файл. В ОС Windows для этого нужно найти SID учетной записи Everyone и добавить ее с разрешением `GENERIC_ALL` в список ACL.

Начнем с поиска SID необходимой записи. Следующий код показывает, как можно найти идентификатор для Everyone:

```
DWORD sidLength = 0;
DWORD sidLengthDomain = 0;
SID_NAME_USE sidType;
LookupAccountName(NULL, L"Гость", NULL, &sidLength,
    NULL, &sidLengthDomain, &sidType);

PSID sidValue = (PSID)new BYTE[sidLength];
LPTSTR wszDomainName = new WCHAR[sidLengthDomain];

if (!LookupAccountName(NULL, L"Гость", sidValue, &sidLength,
        wszDomainName, &sidLengthDomain, &sidType))
{
  printf("Account do not found");
  return 0;
}
```

Все эти функции мы уже знаем, поэтому ничего нового я тут рассказать не могу.

Теперь определяем текущую информацию безопасности файла с помощью `GetNamedSecurityInfo` и ACL-информацию с помощью `GetAclInformation`. Эти функции мы использовали в *разд. 7.2.2*. Можете взять код оттуда, и даже форму главного окна для выбора файла можно оставить старую.

Получив информацию о текущем списке, необходимо выделить память под новый список. Как рассчитать новый размер? Все достаточно просто. У нас есть размер текущего списка в поле `AclBytesInUse` структуры `AclInfo`. Эту структуру мы получили после вызова функции `GetAclInformation`. К этому размеру мы добавляем размер разрешающей структуры `ACCESS_ALLOWED_ACE` и

размер SID идентификатора пользователя, которого мы хотим внести в список:

```
DWORD cbAcl = aclInfo.AclBytesInUse + sizeof(ACCESS_ALLOWED_ACE) +
         GetLengthSid(sidValue);
```

В файле справки из этого результата еще вычитается размер `DWORD`, но я этого не делаю, и у меня пример прекрасно работает. Если будут проблемы, попробуйте изменить строку так:

```
DWORD cbAcl = aclInfo.AclBytesInUse + sizeof(ACCESS_ALLOWED_ACE) +
         GetLengthSid(sidValue) — sezeof(DWORD);
```

Не знаю, зачем нужно вычитать размер числа `DWORD`, возможно, что это действительно нужно, но выделение лишних двух байтов не должно отразиться на надежности кода. Везде, где я тестировал, эти два байта не мешают.

Теперь выделяем память для нового списка ACL и инициализируем его:

```
PACL pNewDACL = NULL;

pNewDACL = (ACL*)LocalAlloc(LPTR, cbAcl);
if (!pNewDACL)
{
  printf("Can't allocate memory");
  return 0;
}

if (!InitializeAcl(pNewDACL, cbAcl, ACL_REVISION))
{
  printf("Can't initialize new DACL");
  return 0;
}
```

Для инициализации используется функция `InitializeAcl`, которая выглядит следующим образом:

```
BOOL WINAPI InitializeAcl(
  __out         PACL pAcl,
  __in          DWORD nAclLength,
  __in          DWORD dwAclRevision
);
```

Функция получает в качестве параметра три значения:

- указатель на выделенную для списка память;
- размер списка;
- ревизия, которая должна быть равна `ACL_REVISION`.

Мы будем использовать последнюю ревизию `ACL_REVISION`, которая равна 2, но если указать 1, то ошибки не должно быть. В старых версиях Windows лучше не экспериментировать, а сразу указывать первую версию, т. е. 1.

Теперь запускаем цикл перебора всех записей и копируем их в новый список, чтобы ничего не потерять:

```
for (DWORD i = 0; i < aclInfo.AceCount; i++)
{
  void *ace;
  if (!GetAce(pDACL, i, &ace))
  {
    printf("Can't get ACE");
    return 0;
  }

  if (!AddAce(pNewDACL, 2, MAXWORD, ace,
            ((ACCESS_ALLOWED_ACE *) ace)->Header.AceSize))
  {
    printf("Add ACE error");
    return 0;
  }
}
```

Цикл прост: сначала пытаемся получить текущую запись. Если неудачно, то переходим на следующий шаг к следующей записи. Если ACE получена, то добавляем ее в новый список с помощью функции `AddAce`, которая в общем виде выглядит так:

```
BOOL WINAPI AddAce(
  __in_out      PACL pAcl,
  __in          DWORD dwAceRevision,
  __in          DWORD dwStartingAceIndex,
  __in          LPVOID pAceList,
  __in          DWORD nAceListLength
);
```

Тут у нас 5 параметров:

- указатель на список, в который нужно добавить запись;
- уже знакомая нам ревизия, которая должна быть равна `ACL_REVISION`;
- позиция, в которую нужно вставить запись. Если указать `0`, то вставка произойдет в начало, а если указать `MAXWORD`, то вставка произойдет в конец списка;

- указатель на один или несколько ACE-записей;
- общий размер добавляемых записей.

Все существующие записи мы уже скопировали в список. Если какую-то из них нужно было удалить, то можно было бы ее не копировать, но тогда нужно было бы правильно пересчитать размер списка.

Теперь нам предстоит добавить новую разрешающую запись, которая не существовала ранее. Для этого используется функция `AddAccessAllowedAce`, которой необходимо передать указатель на список ACL, ревизию, маску прав доступа и идентификатор SID пользователя, которому дается разрешение. В заголовочном файле windows.pas эта функция объявлена следующим образом:

```
BOOL WINAPI AddAccessAllowedAce(
  __in_out      PACL pAcl,
  __in          DWORD dwAceRevision,
  __in          DWORD AccessMask,
  __in          PSID pSid
);
```

Маску прав доступа мы рассматривали в прошлый раз, когда учились читать записи. В нашем случае необходимо разрешить все, так что в третьем параметре указываем флаг `GENERIC_ALL`. SID пользователя Everyone мы уже нашли, добавление новой записи в список ACL будет выглядеть следующим образом:

```
if (!AddAccessAllowedAce(pNewDACL, 2, GENERIC_ALL, sidValue))
{
  printf("AddAccessAllowedAce error");
  return 0;
}
```

Для добавления запрещающей записи необходимо использовать функцию `AddAccessDeniedAce`. У нее параметры такие же, как и у `AddAccessAllowedAce`, разница только в том, что указанные флаги доступа действуют как запрещение. Так как функции очень похожи внешне и одинаковы при вызове, то не будем тратить на рассмотрение запрета место в книге и время.

Все, новый список готов, но он пока еще ни с чем не связан. Необходимо создать новый дескриптор для данного списка. Старый использовать не получится, потому что он занят. Для создания нового дескриптора заведем переменную `pNewSD` типа `PSECURITY_DESCRIPTOR`. Эта переменная является указателем, а значит, требует выделения памяти. Сколько памяти выделить? Вполне достаточно минимума, который равен значению константы `SECURITY_DESCRIPTOR_MIN_LENGTH`. В моем заголовочном файле windows.pas эта констан-

та равна 20. После выделения памяти инициализируем дескриптор с помощью функции InitializeSecurityDescriptor:

```
PSECURITY_DESCRIPTOR pNewSD = PSECURITY_DESCRIPTOR(LocalAlloc(LPTR,
         SECURITY_DESCRIPTOR_MIN_LENGTH));
if (!InitializeSecurityDescriptor(pNewSD,
         SECURITY_DESCRIPTOR_REVISION))
{
  printf("InitializeSecurityDescriptor error");
  return 0;
}
```

Инициализация обязательна, иначе дескриптор будет недоступен, и его нельзя будет связать с ACL-списком.

У функции InitializeSecurityDescriptor два параметра: указатель на переменную дескриптора, и ревизия, которая должна быть равна константе SECURITY_DESCRIPTOR_REVISION. В заголовочном файле эта константа равна единице.

Теперь связываем дескриптор с нашим списком с помощью функции SetSecurityDescriptorDacl. У этой функции четыре параметра:

- указатель на дескриптор, который нужно связать с ACL-списком;
- есть ли ACL-список (true означает, что есть в третьем параметре);
- указатель на ACL-список, с которым происходит связь (true означает, что будет использоваться ACL по умолчанию).

Список доступа есть, он уже связан с дескриптором, осталось только назначить этот дескриптор файлу. Для этого используем функцию SetFileSecurity, у которой три параметра: путь к файлу, тип изменяемой информации (в нашем случае это DACL_SECURITY_INFORMATION) и новый дескриптор. Вот так объявлена эта функция:

```
BOOL WINAPI SetFileSecurity(
  __in          LPCTSTR lpFileName,
  __in          SECURITY_INFORMATION SecurityInformation,
  __in          PSECURITY_DESCRIPTOR pSecurityDescriptor
);
```

Вот и все, новый список назначен файлу, и теперь с этим файлом любой пользователь может делать все что угодно, потому что мы дали разрешение с маской доступа GENERIC_ALL.

Работа с безопасностью окон — достаточно нудное и несколько запутанное занятие. Но упрощать нет смысла, ведь все продумано до мелочей. Другое дело, как эти мелочи реализованы. На данный момент система безопасности Windows уже отлажена и (при правильном подходе) эффективна.

Когда мы программно формировали новый список, то все ACE-записи банально добавлялись в самый конец без сортировки. Если после этого войти в свойства файла и перейти на вкладку **Security**, то ОС Windows предложит отсортировать список. Пример сообщения о том, что список не отсортирован, показан на рис. 7.5. Конечно же, надо согласиться.

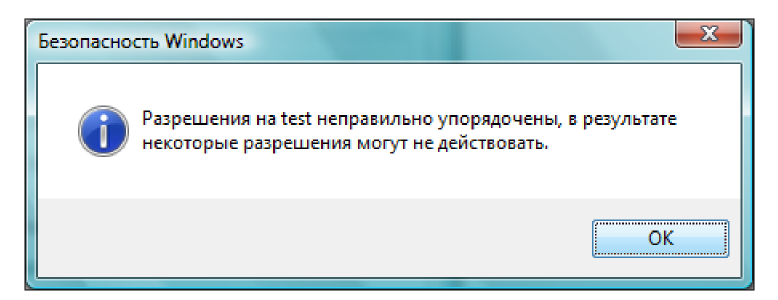


Рис. 7.5. Предупреждение о том, что нужно отсортировать список

Рис. 7.6. Программно добавленный Гость

Я не буду приводить полный код ради экономии страниц, потому что вы сможете его воспроизвести по описанию. На компакт-диске находится полноценный пример, который добавляет разрешающие записи для учетной записи Гость. На рис. 7.6 можно увидеть результат, в котором явно добавлена учетная запись. Поверьте, что я ее добавил программно, или запустите пример в своей системе и убедитесь сами.

Точный алгоритм сортировки я не нашел. В данном случае мое разрешение гостевой записи система поставила на первое место.



Исходный код примера, описанного в этом разделе, вы можете найти на компакт-диске в каталоге \Demo\Chapter7\DaclControl.

Заключение

В этой книге я привел достаточно много интересных примеров, но нельзя сказать, что это позволит постигнуть все приемы хакеров — искусство, которому учатся всю жизнь. Информационная и компьютерная сферы жизни развиваются настолько динамично, что поспеть сразу за всем просто невозможно.

Однажды я целый год был настолько загружен программированием, что не успевал заниматься самообучением, и когда оглянулся вокруг, то увидел, что сильно отстал от жизни. Я все еще писал программы под MS-DOS, в то время как другие уже осваивали Windows. Я понял, что пора менять работу, иначе из-за такой отсталости мои знания станут никому не нужными. Уволившись, мне пришлось потратить три месяца на изучение хотя бы базовых возможностей, которые предоставляет ОС Windows, чтобы начать писать коммерческие проекты и найти новую работу, соответствующую моим знаниям, и где можно было самосовершенствоваться.

Меня часто спрашивают, что можно еще почитать, чтобы стать настоящим хакером или повысить уровень знаний? На этот вопрос я отвечаю: "Абсолютно все!!!" Я читаю любые документы, файлы помощи, статьи, которые попадаются мне на глаза. Я каждый месяц покупаю книги, чтобы читать в любое свободное время. Пусть я большую часть уже в этой книге знаю, но иногда маленькие решения, которые находят другие люди, очень сильно помогают, и это стоит потраченных денег. Специализированной литературы практически нет, но заранее никогда не известно, в какой книге встретишь что-то новое и интересное, что позволит сделать нечто совершенное и непохожее на другие проекты.

Но даже если тратить достаточно времени на обучение, нельзя гарантировать, что именно вы станете настоящим хакером. Во время учебы в институте я работал за компьютером по 10–16 часов в сутки и успевал практически все, что

хотелось. Сейчас я уже женат, и у меня двое детей, и хотя я стараюсь работать не менее 9 часов, успеть везде уже невозможно. Хочется делать больше и постоянно следить за всем, что появляется в мире информационных технологий, но это нереально.

В любом случае, я надеюсь, что эта книга помогла вам найти ответы на интересующие вопросы. Если у вас возникли какие-то проблемы, есть комментарии или просто пожелания, то жду вас на своем сайте **http://www.flenov.info**. Я всегда готов помочь по мере своих возможностей. Если долго не откликаюсь на ваше письмо, то это не значит, что я забыл, просто каждый день мне приходит множество писем, и ответить всем сразу я не в состоянии. Но я стремлюсь отвечать оперативно.

ПРИЛОЖЕНИЕ

Описание компакт-диска

На прилагаемом к книге компакт-диске находятся следующие материалы (табл. П1).

Таблица П1. *Содержание компакт-диска*

Папки	**Описание**
\Doc	Дополнительная документация
\Demo	Исходные коды простых программ, чтобы вы могли ознакомиться с реальными приложениями. Их немного, но посмотреть стоит
\Demo\Chapter1	Исходные коды к главе 1 "Оптимизация"
\Demo\Chapter2	Исходные коды к главе 2 "Простые шутки"
\Demo\Chapter3	Исходные коды к главе 3 "Система"
\Demo\Chapter4	Исходные коды к главе 4 "Работа с сетью"
\Demo\Chapter5	Исходные коды к главе 5 "Работа с железом"
\Demo\Chapter6	Исходные коды к главе 6 "Полезные примеры"
\Demo\Chapter7	Исходные коды к главе 7 "Система безопасности"
\Soft	Демонстрационные программы
\Sources	Исходные коды

Список литературы и ресурсы Интернета

1. Фленов М. Е. Программирование в Delphi глазами хакера. — СПб.: БХВ-Петербург, 2003.
2. Фленов М. Е. Библия Delphi. — СПб.: БХВ-Петербург, 2004.
3. Рубрика "Кодинг" журнала "Хакер" (Гэймлэнд) — всегда содержит полезные советы и примеры на C++ и Delphi.
4. **http://www.flenov.info** — сайт автора книги.
5. **http://www.hackishcode.com** — сайт, посвященный программированию на C/C++/C#, с полезной документацией и исходными кодами.

Предметный указатель

О

П

Р

С

Т

У

Ф





Э

